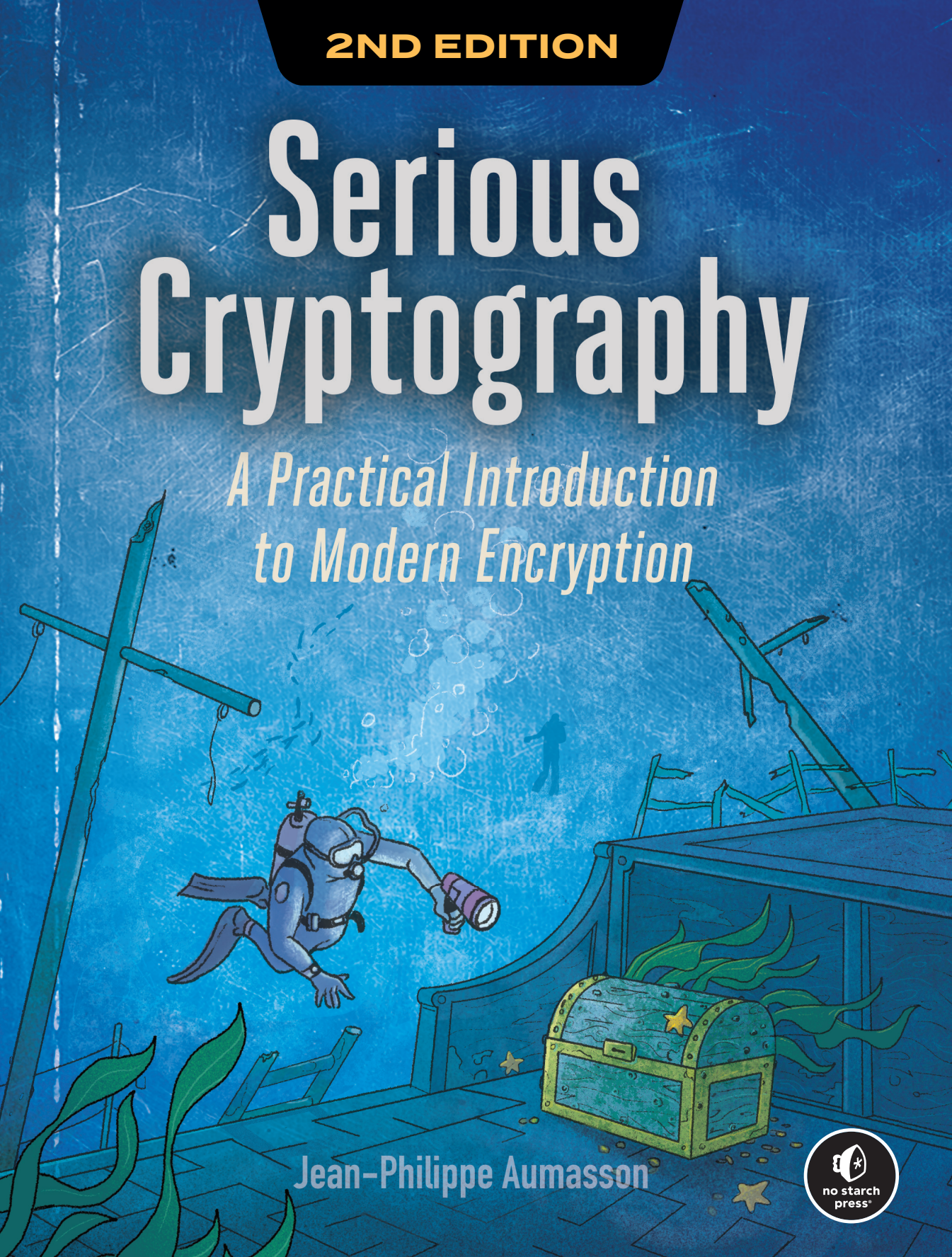
2ND EDITION

Serious Cryptography

A Practical Introduction to Modern Encryption



Jean-Philippe Aumasson

no starch press®

PRAISE FOR *SERIOUS CRYPTOGRAPHY*

"Fills [the need for an accessible and readable resource on cryptography], taking the reader on a journey through different cryptographic tools and how to use them, as well as the important 'what can go wrong' sections that fill the book. . . . *Serious Cryptography* is an enjoyable introduction to the field, and one that comes highly recommended."

—YEHUDA LINDELL, HEAD OF CRYPTOGRAPHY AT COINBASE

"A properly serious introduction; it describes the essential concepts in a clear and concise prose. It does not aim at vacuous entertainment of the reader, but at imparting knowledge. This is the kind of 'first book' that you'd keep referring to because it has the right structure on which one can build a thorough understanding of cryptography."

—THOMAS PORNIN, TECHNICAL VICE PRESIDENT, NCC GROUP

"Like having a wise friend explain all of cryptography's mysteries without making your head spin. The book stands out for its practical approach, which provides real-world applications and insights. It is considered mandatory reading for security engineers undergoing onboarding within my team, providing an essential foundation for their professional development."

—ANASTASIIA VOITOVA, HEAD OF SECURITY ENGINEERING AT COSSACK LABS

"A superb introduction to modern encryption and cryptography. For those looking to quickly get up to speed on the topics, this makes for an excellent go-to guide."

—BEN ROTHKE, RSA CONFERENCE

"It's really a love letter to cryptography."

—NADIM KOBEISSI

"Aumasson successfully ensures that the reader has a strong understanding of cryptography's core ideas. . . . *Serious Cryptography* is a must read for anyone wanting to enter cryptographic engineering."

—GREGORY PARFITT, *INFOSECURITY MAGAZINE*

SERIOUS CRYPTOGRAPHY

2nd Edition

A Practical Introduction to Modern Encryption

by Jean-Philippe Aumasson

no starch press®

San Francisco

SERIOUS CRYPTOGRAPHY, 2ND EDITION. Copyright © 2025 by Jean-Philippe Aumasson.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Second printing
30 29 28 27 26 2 3 4 5 6

ISBN-13: 978-1-7185-0384-7 (print)
ISBN-13: 978-1-7185-0385-4 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Miles Bond
Developmental Editor: Eva Morrow
Cover Illustrator: Rick Reese
Interior Design: Octopod Studios
Technical Reviewer: Pascal Junod
Copyeditor: Kim Wimpsett
Proofreader: James Brook

Library of Congress Control Number: 2017940486

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

[E]

About the Author

JP Aumasson has been developing, researching, and breaking cryptography since 2006. He's at the origin of the widely used crypto algorithms BLAKE2, BLAKE3, and SipHash, and he organized the Password Hashing Competition that led to the Argon2 standard. JP's career spans academia, where he earned a PhD from EPFL; the pay-TV security industry; and roles as a security consultant for both private- and public-sector clients. He's currently co-founder and chief security officer at Taurus SA, a global fintech company based in Geneva that assists financial institutions in safeguarding and managing their digital assets.

About the Technical Reviewer

Pascal Junod is a cryptographer with more than 20 years of academic and industrial experience in designing, implementing, auditing, breaking, writing, and speaking about real-world cryptographic and cybersecurity systems. His main fields of interest include industrial cryptography, software protection, and reverse engineering. He also enjoys trail running, playing the accordion, and taking care of his family.

BRIEF CONTENTS

Foreword to the First Edition xvii

Acknowledgments xix

Introduction xxi

List of Abbreviations xxv

PART I: FUNDAMENTALS 1

Chapter 1: Encryption 3

Chapter 2: Randomness 25

Chapter 3: Cryptographic Security 43

PART II: SYMMETRIC CRYPTO 59

Chapter 4: Block Ciphers 61

Chapter 5: Stream Ciphers 87

Chapter 6: Hash Functions 115

Chapter 7: Keyed Hashing 139

Chapter 8: Authenticated Encryption 157

PART III: ASYMMETRIC CRYPTO 175

Chapter 9: Hard Problems 177

Chapter 10: RSA 195

Chapter 11: Diffie–Hellman 215

Chapter 12: Elliptic Curves 231

PART IV: APPLICATIONS 253

Chapter 13: TLS 255

Chapter 14: Quantum and Post-Quantum 273

Chapter 15: Cryptocurrency Cryptography 295

Index 333

CONTENTS IN DETAIL

FOREWORD TO THE FIRST EDITION **xvii**

ACKNOWLEDGMENTS **xix**

INTRODUCTION **xxi**

This Book's Approach . . . xxii
Who This Book Is For . . . xxii
How This Book Is Organized . . . xxiii
On the Second Edition . . . xxiv

LIST OF ABBREVIATIONS **xxv**

PART I: FUNDAMENTALS **1**

1
ENCRYPTION **3**

The Basics . . . 4
Classical Ciphers . . . 4
The Caesar Cipher . . . 4
The Vigenère Cipher . . . 5
How Ciphers Work . . . 6
The Permutation . . . 7
The Mode of Operation . . . 8
Why Classical Ciphers Are Insecure . . . 8
The Perfect Cipher: The One-Time Pad . . . 9
Encryption and Decryption . . . 9
Why Is the One-Time Pad Secure? . . . 10
Encryption Security . . . 12
Attack Models . . . 12
Security Goals . . . 15
Security Notions . . . 15
Asymmetric Encryption . . . 18
When Ciphers Do More Than Encryption . . . 18
Authenticated Encryption . . . 18
Format-Preserving Encryption . . . 19
Fully Homomorphic Encryption . . . 20
Searchable Encryption . . . 20
Tweakable Encryption . . . 20

How Things Can Go Wrong 21
Weak Cipher 21
Wrong Model 22
Further Reading 22

2
RANDOMNESS **25**

Random or Nonrandom? 25
Randomness as a Probability Distribution 26
Entropy: A Measure of Uncertainty 27
Random and Pseudorandom Number Generators 28
How PRNGs Work 29
Security Concerns 30
The PRNG Fortuna 30
Cryptographic vs. Noncryptographic PRNGs 32
The Uselessness of Statistical Tests 33
Real-World PRNGs 34
Random Bits in Linux 34
The CryptGenRandom() Function in Windows 37
A Hardware-Based PRNG: Intel Secure Key 38
How Things Can Go Wrong 38
Poor Entropy Sources 39
Insufficient Entropy at Boot Time 39
Noncryptographic PRNG 40
Sampling Bug with Strong Randomness 41
Further Reading 41

3
CRYPTOGRAPHIC SECURITY **43**

Defining the Impossible 44
Security in Theory: Unconditional Security 44
Security in Practice: Computational Security 44
Quantifying Security 46
Measuring Security in Bits 46
Calculating the Full Attack Cost 47
Choosing and Evaluating Security Levels 49
Achieving Security 50
Provable Security 50
Heuristic Security 52
Generating Keys 53
Symmetric Keys 53
Asymmetric Keys 54
Protecting Keys 55
How Things Can Go Wrong 56
Incorrect Security Proof 56
Short Keys for Legacy Support 56
Further Reading 57

PART II: SYMMETRIC CRYPTO 59

4
BLOCK CIPHERS 61

What Is a Block Cipher? 62
Security Goals 62
Block Size 62
The Codebook Attack 63
How to Construct Block Ciphers 63
A Block Cipher's Rounds 64
The Slide Attack and Round Keys 64
Substitution–Permutation Networks 65
Feistel Schemes 66
The Advanced Encryption Standard 67
AES Internals 67
AES in Action 70
How to Implement AES 71
Table-Based Implementations 71
Native Instructions 72
AES Security 73
Modes of Operation 74
Electronic Codebook Mode 74
Cipher Block Chaining Mode 76
Message Encryption in CBC Mode 78
Counter Mode 80
How Things Can Go Wrong 82
Meet-in-the-Middle Attacks 82
Padding Oracle Attacks 83
Further Reading 85

5
STREAM CIPHERS 87

How Stream Ciphers Work 88
Hardware-Oriented Stream Ciphers 89
Feedback Shift Registers 90
Grain-128a 97
A5/1 98
Software-Oriented Stream Ciphers 101
RC4 102
Salsa20 106
How Things Can Go Wrong 111
Nonce Reuse 111
Broken RC4 Implementation 111
Weak Ciphers Baked into Hardware 113
Further Reading 113

6
HASH FUNCTIONS 115

Secure Hash Functions 116
Unpredictability Again 117
Preimage Resistance 117

Collision Resistance . . . 119
How to Find Collisions . . . 120
How to Build Hash Functions . . . 122
Compression-Based Hash Functions . . . 122
Permutation-Based Hash Functions . . . 125
The SHA Family of Hash Functions . . . 126
SHA-1 . . . 127
SHA-2 . . . 129
The SHA-3 Competition . . . 131
Keccak (SHA-3) . . . 132
The BLAKE2 and BLAKE3 Hash Functions . . . 133
How Things Can Go Wrong . . . 135
The Length-Extension Attack . . . 135
Fooling Proof-of-Storage Protocols . . . 136
Further Reading . . . 137

7
KEYED HASHING **139**

Message Authentication Codes . . . 140
MACs in Secure Communication . . . 140
Forgery and Chosen-Message Attacks . . . 140
Replay Attacks . . . 141
Pseudorandom Functions . . . 141
PRF Security . . . 141
PRFs Are Stronger Than MACs . . . 141
How to Create Keyed Hashes from Unkeyed Hashes . . . 142
The Secret-Prefix Construction . . . 142
The Secret-Suffix Construction . . . 143
The HMAC Construction . . . 144
A Generic Attack Against Hash-Based MACs . . . 145
How to Create Keyed Hashes from Block Ciphers . . . 146
Breaking CBC-MAC . . . 146
Fixing CBC-MAC . . . 146
Dedicated MAC Designs . . . 148
Poly1305 . . . 148
SipHash . . . 151
How Things Can Go Wrong . . . 153
Timing Attacks on MAC Verification . . . 153
When Sponges Leak . . . 155
Further Reading . . . 155

8
AUTHENTICATED ENCRYPTION **157**

Authenticated Encryption Using MACs . . . 158
Encrypt-and-MAC Approach . . . 158
MAC-Then-Encrypt Composition . . . 159
Encrypt-Then-MAC Composition . . . 159
Authenticated Ciphers . . . 160
Authenticated Encryption with Associated Data . . . 160
Predictability and Nonces . . . 161
Criteria for a Good Authenticated Cipher . . . 162

The AES-GCM Authenticated Cipher Standard . . . 164
GCM Internals . . . 164
GCM Security . . . 166
GCM Efficiency . . . 166
The OCB Authenticated Cipher Mode . . . 167
OCB Internals . . . 167
OCB Security . . . 168
OCB Efficiency . . . 168
The SIV Authenticated Cipher Mode . . . 169
Permutation-Based AEAD . . . 169
How Things Can Go Wrong . . . 171
AES-GCM and Weak Hash Keys . . . 171
AES-GCM and Small Tags . . . 173
Further Reading . . . 173

PART III: ASYMMETRIC CRYPTO 175

9
HARD PROBLEMS 177

Computational Hardness . . . 178
Running Time . . . 178
Polynomial vs. Superpolynomial Time . . . 180
Complexity Classes . . . 182
Nondeterministic Polynomial Time . . . 182
NP-Complete Problems . . . 183
The P vs. NP Problem . . . 185
The Factoring Problem . . . 186
Factoring Large Numbers . . . 187
Factoring Is Probably Not NP-Hard . . . 188
The Discrete Logarithm Problem . . . 189
Groups . . . 189
The Hard Thing . . . 190
How Things Can Go Wrong . . . 191
When Factoring Is Easy . . . 191
Small Hard Problems Aren't Hard . . . 192
Further Reading . . . 193

10
RSA 195

The Math Behind RSA . . . 196
The RSA Trapdoor Permutation . . . 197
RSA Key Generation and Security . . . 198
Encrypting with RSA . . . 199
Textbook RSA Encryption's Malleability . . . 199
Strong RSA Encryption with OAEP . . . 200
Signing with RSA . . . 202
Textbook RSA Signatures . . . 203
The PSS Signature Standard . . . 203
Full Domain Hash Signatures . . . 205

RSA Implementations 205
A Fast Exponentiation Algorithm 206
Small Exponents for Faster Public-Key Operations 208
The Chinese Remainder Theorem 210
How Things Can Go Wrong 211
The Bellcore Attack on RSA-CRT 211
Shared Private Exponents or Moduli 212
Further Reading 213

11
DIFFIE–HELLMAN **215**

The Diffie–Hellman Function 216
The Diffie–Hellman Problems 218
The Computational Problem 218
The Decisional Problem 218
Variants of Diffie–Hellman 219
Key Agreement Protocols 219
Non-DH Key Agreement. 219
Attack Models for Key Agreement Protocols 221
Performance. 222
Diffie–Hellman Protocols 223
Anonymous Diffie–Hellman. 223
Authenticated Diffie–Hellman 224
Menezes–Qu–Vanstone 227
How Things Can Go Wrong 228
Not Hashing the Shared Secret. 228
Anonymous Diffie–Hellman from TLS 1.0 229
Unsafe Group Parameters 229
Further Reading 230

12
ELLIPTIC CURVES **231**

What Is an Elliptic Curve?. 232
Elliptic Curves Over Integers. 233
The Addition Law 235
Elliptic Curve Groups. 239
The ECDLP Problem 240
Diffie–Hellman Key Agreement over Elliptic Curves. 241
Signing with Elliptic Curves 241
ECDSA Signature Generation. 242
ECDSA Signature Verification 242
ECDSA vs. RSA Signatures. 243
EdDSA and Ed25519 244
Encrypting with Elliptic Curves. 246
Choosing a Curve 246
NIST Curves. 247
Curve25519 248
Other Curves 248

How Things Can Go Wrong 249
ECDSA with Bad Randomness 249
Invalid Curve Attacks 249
Incompatible Ed25519 Validation Rules 250
Further Reading 251

PART IV: APPLICATIONS 253

13
TLS 255

Target Applications and Requirements 256
The TLS Protocol Suite 256
The TLS and SSL Families of Protocols 257
TLS in a Nutshell 257
Certificates and Certificate Authorities 258
The Record Protocol 262
The TLS Handshake Protocol 263
TLS 1.3 Cryptographic Algorithms 265
TLS 1.3 Improvements over TLS 1.2 266
Downgrade Protection 266
Single Round-Trip Handshake 267
Session Resumption 267
The Strengths of TLS Security 268
Authentication 268
Forward Secrecy 268
How Things Can Go Wrong 269
Compromised Certificate Authority 269
Compromised Server 269
Compromised Client 270
Bugs in Implementations 270
Further Reading 271

14
QUANTUM AND POST-QUANTUM 273

How Quantum Computers Work 274
Quantum Bits 274
Quantum Gates 277
Quantum Speedup 279
Exponential Speedup and Simon's Problem 280
The Threat of Shor's Algorithm 281
Grover's Algorithm 282
Why Is It So Hard to Build a Quantum Computer? 284
Post-Quantum Cryptographic Algorithms 285
Code-Based Cryptography 285
Lattice-Based Cryptography 286
Multivariate Cryptography 287
Hash-Based Cryptography 288
The NIST Standards 289

How Things Can Go Wrong 291
Unclear Security Level 291
The Eventual Existence of Large Quantum Computers. 291
Implementation Issues 292
Further Reading 293

15
CRYPTOCURRENCY CRYPTOGRAPHY 295

Hashing Applications 296
Merkle Trees. 297
Proof of Work. 300
Hierarchical Key Derivation 301
Algebraic Hash Functions. 302
How Things Can Go Wrong. 304
Multisignature Protocols 306
Multiple Multiparty Signatures 306
Schnorr Signature Protocols 307
How Things Can Go Wrong. 309
Safer Schnorr Multisignatures 310
Aggregate Signature Protocols. 311
Pairings 312
BLS Signatures 312
How Things Can Go Wrong. 314
Threshold Signature Protocols 316
Use Cases 316
Security Model 317
Secret-Sharing Techniques 319
The Trivial Case 320
The Simple Case. 320
The Hard Case 321
How Things Can Go Wrong. 321
Zero-Knowledge Proofs. 323
Security Model 324
Schnorr's Protocol 325
Noninteractive Proofs 326
zkSNARKs 327
From Statements to Proofs. 328
How Things Can Go Wrong. 329
Really Serious Crypto 330

INDEX 333

FOREWORD TO THE FIRST EDITION

If you've read a book or two on computer security, you may have encountered a common perspective on the field of cryptography. "Cryptography," they say, "is the strongest link in the chain." Strong praise indeed, but it's also somewhat dismissive. If cryptography is in fact the strongest part of your system, why invest time improving it when there are so many other areas of the system that will benefit more from your attention?

If there's one thing that I hope you take away from this book, it's that this view of cryptography is idealized; it's largely a myth. Cryptography *in theory* is strong, but cryptography in practice is as prone to failure as any other aspect of a security system. This is particularly true when cryptographic implementations are developed by nonexperts without sufficient care or experience, as is the case with many cryptographic systems deployed today. And it gets worse: when cryptographic implementations fail, they often do so in uniquely spectacular ways.

But why should you care, and why this book?

When I began working in the field of applied cryptography nearly two decades ago, the information available to software developers was often piecemeal and outdated. Cryptographers developed algorithms and protocols, and cryptographic engineers implemented them to create opaque, poorly documented cryptographic libraries designed mainly for other experts. There was—and there has been—a huge divide between those who

know and understand cryptographic algorithms and those who use them (or ignore them at their peril). There are a few decent textbooks on the market, but even fewer have provided useful tools for the practitioner.

The results have not been pretty. I'm talking about compromises with labels like "CVE" and "Severity: High," and in a few alarming cases, attacks on slide decks marked "TOP SECRET." You may be familiar with some of the more famous examples if only because they've affected systems that you rely on. Many of these problems occur because cryptography is subtle and mathematically elegant, and because cryptographic experts have failed to share their knowledge with the engineers who actually write the software.

Thankfully, this has begun to change, and this book is a symptom of that change.

Serious Cryptography was written by one of the foremost experts in applied cryptography, but it's not targeted at other experts. Nor, for that matter, is it intended as a superficial overview of the field. On the contrary, it contains a thorough and up-to-date discussion of cryptographic engineering, designed to help practitioners who plan to work in this field do better. In these pages, you'll learn not only how cryptographic algorithms work but also how to use them in real systems.

The book begins with an exploration of many of the key cryptographic primitives, including basic algorithms like block ciphers, public encryption schemes, hash functions, and random number generators. Each chapter provides working examples of how the algorithms work and what you should or should *not* do. Final chapters cover advanced subjects such as TLS, as well as the future of cryptography—what to do after quantum computers arrive to complicate our lives.

While no single book can solve all our problems, a bit of knowledge can go a long way. This book contains plenty of knowledge. Perhaps enough to make real, deployed cryptography live up to the high expectations that so many have of it.

Happy reading.

Matthew D. Green
Associate Professor of Computer Science
Information Security Institute
Johns Hopkins University

ACKNOWLEDGMENTS

First Edition

I'd like to thank Jan, Annie, and the rest of the No Starch staff who contributed to this book, especially Bill for believing in this project from the get-go, for his patience digesting difficult topics, and for turning my clumsy drafts into readable pages. I am also thankful to Laurel for making the book look so nice and for handling my many correction requests.

On the technical side, the book would contain many more errors and inaccuracies without the help of the following people: Jon Callas, Bill Cox, Niels Ferguson, Philipp Jovanovic, Samuel Neves, David Reid, Phillip Rogaway, and Erik Tews, as well as all readers of the Early Access version who reported errors. Finally, thanks to Matt Green for writing the foreword.

I'd also like to thank my previous employer, Kudelski Security, for allowing me time to work on this book. Finally, I offer my deepest thanks to Alexandra and Melina for their support and patience.

Lausanne, 05/17/2017 (three prime numbers)

Second Edition

I'd like to thank all the readers of the first edition, in particular those who reached out to share feedback and help improve the book. This second edition greatly benefited from the meticulous editing by No Starch Press (thanks to Eva, Jill, Kim, and Miles) and the careful technical review by Pascal Junod. Needless to say, the shortcomings of this book are my own.

Lausanne, 04/14/2024

INTRODUCTION

I wrote this book to be the one I wish I had when I started learning cryptography. In 2005, I was studying for my master's degree near Paris, and I eagerly registered for the upcoming semester's crypto class. Unfortunately, the class was canceled because too few students had registered. "Crypto is too hard," the students argued, instead enrolling en masse in the computer graphics and database classes.

I've heard "crypto is hard" dozens of times since then. But is it really *that* hard? To play an instrument, master a programming language, or put the applications of any field into practice, you need to learn some concepts and symbols, but doing so doesn't take a PhD. The same applies to becoming a competent cryptographer. Perhaps crypto is perceived as hard because cryptographers haven't done a good job of teaching it.

I also wrote this book because cryptography has expanded into a multidisciplinary field. To do anything useful and relevant in crypto, you need to understand the concepts *around* crypto: how networks and computers work, what users and systems need, and how attackers can abuse algorithms and their implementations. In other words, you need a connection to reality.

This Book's Approach

The initial title of this book was *Crypto for Real* to stress the practice-oriented, real-world, no-nonsense approach I follow. I wanted to make cryptography approachable not by dumbing it down but by tying it to real applications. I provide source code examples and describe real bugs and horror stories.

Along with a clear connection to reality, other cornerstones of this book are its simplicity and its modernity. I focus on simplicity in form more than in substance: I present nontrivial concepts without the dull mathematical formalism. Instead, I attempt to impart an understanding of cryptography's core ideas, which are more important than remembering a bunch of equations. To ensure the book's modernity, I cover the latest developments and applications of cryptography, such as TLS 1.3 and post-quantum cryptography. I don't discuss the details of obsolete or insecure algorithms such as DES or MD5. An exception to this is RC4, but it's included only to explain how weak it is and to show how a stream cipher of its kind works.

Serious Cryptography isn't a guide to crypto software, nor is it a compendium of technical specifications—stuff that you'll easily find online. Instead, its foremost goal is to get you excited about cryptography and teach you its fundamental concepts along the way.

Who This Book Is For

While writing, I often imagined the reader as a developer who'd been exposed to cryptography but still felt clueless and frustrated after reading abstruse textbooks and research papers. Developers often need—and want—a better grasp of crypto to avoid unfortunate design choices, and I hope this book helps.

If you aren't a developer, don't worry! The book doesn't require coding skills and is accessible to anyone who understands the basics of computer science and high school math (notions of probabilities, modular arithmetic, and so on).

This book can nonetheless be intimidating, and despite its relative accessibility, it requires some effort to get the most out of it. I like the mountaineering analogy: the author paves the way, providing you with ropes and ice axes to facilitate your work, but you make the ascent yourself. Learning the concepts in this book takes effort but is rewarding.

How This Book Is Organized

The book has 15 chapters, loosely split into four parts. The chapters are mostly independent from one another, except for Chapter 9, which lays the foundations for the three subsequent chapters. I recommend reading the first three chapters before anything else.

Part I: Fundamentals

Chapter 1: Encryption Introduces the notion of secure encryption, from weak pen-and-paper ciphers to strong, randomized encryption

Chapter 2: Randomness Describes how a pseudorandom generator works, what it takes for one to be secure, and how to use one securely

Chapter 3: Cryptographic Security Discusses theoretical and practical notions of security and compares provable security with probable security

Part II: Symmetric Crypto

Chapter 4: Block Ciphers Deals with ciphers that process messages block per block, focusing on the most famous one, the Advanced Encryption Standard (AES)

Chapter 5: Stream Ciphers Presents ciphers that produce a stream of random-looking bits that are XORed with messages to be encrypted

Chapter 6: Hash Functions Discusses the only algorithms that don't work with a secret key, which turn out to be the most ubiquitous crypto building blocks

Chapter 7: Keyed Hashing Explains what happens if you combine a hash function with a secret key and how this serves to authenticate messages

Chapter 8: Authenticated Encryption Shows how some algorithms can both encrypt and authenticate a message, with examples such as the standard AES-GCM

Part III: Asymmetric Crypto

Chapter 9: Hard Problems Lays out the fundamental concepts behind public-key encryption, using notions from computational complexity

Chapter 10: RSA Leverages the factoring problem in order to build secure encryption and signature schemes with a simple arithmetic operation

Chapter 11: Diffie–Hellman Extends asymmetric cryptography to the notion of key agreement, wherein two parties establish a secret value using only nonsecret values

Chapter 12: Elliptic Curves Provides a gentle introduction to elliptic curve cryptography, which is the fastest kind of asymmetric cryptography

Part IV: Applications

Chapter 13: TLS Focuses on Transport Layer Security (TLS), arguably the most important protocol in network security

Chapter 14: Quantum and Post-Quantum Presents the concepts of quantum computing and post-quantum cryptography

Chapter 15: Cryptocurrency Cryptography Concludes with an overview of advanced cryptographic schemes found in blockchain applications

On the Second Edition

This second edition of *Serious Cryptography* comes seven years after the first edition. Since then, cryptography has experienced significant changes. Nowadays, the term *crypto* often conjures thoughts of blockchain, Bitcoin, and other cryptocurrencies, rather than cryptography itself. Despite the debatable societal benefits of these technologies, their undeniable influence on the advancement of cryptography research and engineering can't be overlooked. Recognizing this, I've written Chapter 15, "Cryptocurrency Cryptography," which delves into fascinating cryptographic techniques employed in blockchain applications, representing some of the most intriguing advancements in the field of cryptography.

I've made substantial changes to each chapter, updating the text with respect to new cryptography developments and improving the text's clarity and conciseness. Among the most significant additions: Chapter 2's discussion of Linux kernel randomness was updated to describe the new behavior of the */dev/random* and */dev/urandom* interfaces, Chapter 12 features a new section on the EdDSA and Ed25519 signature schemes, and Chapter 14 presents NIST's Post-Quantum Cryptography Standardization project.

ABBREVIATIONS

AE	authenticated encryption
AEAD	authenticated encryption with associated data
AES	Advanced Encryption Standard
AES-NI	AES native instructions
AKA	authenticated key agreement
API	application programming interface
ARX	add-rotate-XOR
ASIC	application-specific integrated circuit
BLS	Barreto–Lynn–Scott
BLS	Boneh–Lynn–Shacham
CA	certificate authority
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
CBC	cipher block chaining
CCA	chosen-ciphertext attackers
CDH	computational Diffie–Hellman
CMAC	cipher-based MAC
COA	ciphertext-only attackers
CPA	chosen-plaintext attackers
CRT	Chinese remainder theorem
CTR	counter mode
CVP	closest vector problem
DDH	decisional Diffie–Hellman
DES	Data Encryption Standard
DH	Diffie–Hellman
DLP	discrete logarithm problem
DRBG	deterministic random bit generator
ECB	electronic codebook
ECC	elliptic curve cryptography
ECDH	elliptic curve Diffie–Hellman
ECDLP	elliptic-curve discrete logarithm problem
ECDSA	elliptic-curve digital signature algorithm

FDH	Full Domain Hash
FHE	fully homomorphic encryption
FIPS	Federal Information Processing Standards
FPE	format-preserving encryption
FPGA	field-programmable gate array
FSR	feedback shift register
GCD	greatest common divisor
GCM	Galois Counter Mode
GNFS	general number field sieve
HKDF	HMAC-based key derivation function
HMAC	hash-based message authentication code
HTTPS	HTTP Secure
IND	indistinguishability
IP	Internet Protocol
IV	initial value
KDF	key derivation function
KPA	known-plaintext attackers
LFSR	linear feedback shift register
LSB	least significant bit
LWE	learning with errors
MAC	message authentication code
MD	message digest
MitM	meet-in-the-middle
MPC	multiparty computation
MQ	multivariate quadratics
MQV	Menezes–Qu–Vanstone
MSB	most significant bit
MT	Mersenne Twister
NFSR	nonlinear feedback shift register
NIST	National Institute of Standards and Technology
NM	nonmalleability
NP	nondeterministic polynomial-time
OAEP	Optimal Asymmetric Encryption Padding
OCB	offset codebook
P	polynomial time
PLD	programmable logic device
PoW	proof of work
PRF	pseudorandom function
PRNG	pseudorandom number generator
PRP	pseudorandom permutation
PSK	preshared key
PSS	Probabilistic Signature Scheme
QR	quarter-round
QRNG	quantum random number generator
RFC	request for comments
RNG	random number generator
RSA	Rivest–Shamir–Adleman
SHA	Secure Hash Algorithm
SIS	short integer solution
SIV	synthetic IV
SNARK	succinct noninteractive argument of knowledge
SPN	substitution–permutation network
SSH	Secure Shell
SSL	Secure Sockets Layer
TE	tweakable encryption
TLS	Transport Layer Security
TMTO	time-memory trade-off
UDP	User Datagram Protocol
UH	universal hash
WEP	Wireless Encrypted Protocol
WOTS	Winternitz one-time signature
XOR	exclusive OR
ZKP	zero-knowledge proof

PART I

FUNDAMENTALS

1

ENCRYPTION

Encryption is the principal application of cryptography; it makes data incomprehensible to ensure its *confidentiality*. Encryption uses an algorithm called a *cipher* and a secret value called the *key*. If you don't know the secret key, you can't decrypt, nor can you learn any bit of information on the encrypted message—and neither can any attacker.

This chapter focuses on symmetric encryption, which is the simplest kind of encryption. In *symmetric encryption*, the decryption key is the same as the encryption key (unlike *asymmetric encryption*, or *public-key encryption*, in which the keys are different). You'll start by learning about the weakest forms of symmetric encryption, classical ciphers that are secure against only the most illiterate attacker, and then we'll move on to the strongest forms that are secure forever.

The Basics

When encrypting a message, *plaintext* refers to the unencrypted message and *ciphertext* to the encrypted message. A cipher is therefore composed of two functions: *encryption* turns a plaintext into a ciphertext, and *decryption* turns a ciphertext back into a plaintext. But we'll often say "cipher" when we actually mean "encryption." For example, Figure 1-1 shows a cipher, **E**, represented as a box taking as input a plaintext, P, and a key, K, and producing a ciphertext, C, as output. I'll write this relation as $C = \mathbf{E}(K, P)$. Similarly, when the cipher is in decryption mode, I'll write $\mathbf{D}(K, C)$.

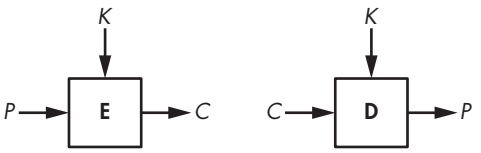


Figure 1-1: Basic encryption and decryption

NOTE *For some ciphers, the ciphertext is the same size as the plaintext; for others, the ciphertext is slightly longer. However, ciphertexts can never be shorter than plaintexts.*

Classical Ciphers

Classical ciphers predate computers and therefore work on letters rather than on bits, making them much simpler than a modern cipher like the Data Encryption Standard. For example, in ancient Rome or during World War I, you couldn't use a computer chip's power to scramble a message; you had to do everything with only pen and paper. There are many classical ciphers, but the most famous are the Caesar cipher and Vigenère cipher.

The Caesar Cipher

The Caesar cipher is so named because the Roman historian Suetonius reported that Julius Caesar used it. It encrypts a message by shifting each of the letters down three positions in the alphabet, wrapping back around to *A* if the shift reaches *Z*. For example, *ZOO* encrypts to *CRR*, *FDHVDU* decrypts to *CAESAR*, and so on, as shown in Figure 1-2. There's nothing special about the value 3; it's just easier to compute in one's head than 11 or 23.

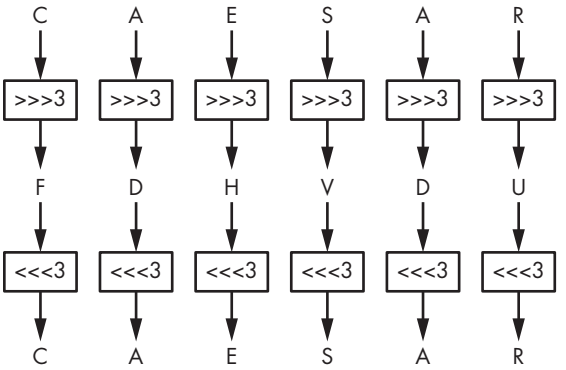


Figure 1-2: The Caesar cipher

The Caesar cipher is super easy to break: to decrypt a given ciphertext, simply shift the letters three positions back to retrieve the plaintext. That said, the Caesar cipher may have been strong enough during the time of Crassus and Cicero. Because no secret key is involved (it's always 3), users of Caesar's cipher assumed that attackers were illiterate or too uneducated to figure it out—an assumption that's much less realistic today. (In fact, in 2006, the Italian police arrested a mafia boss after decrypting messages written on small scraps of paper that were encrypted using a variant of the Caesar cipher: *ABC* was encrypted to 456 instead of *DEF*, for example.)

Could the Caesar cipher be made more secure? You might imagine a version that uses a secret shift value instead of always using 3, but that wouldn't help much because an attacker could try all 25 possible shift values until the decrypted message makes sense.

The Vigenère Cipher

It took about 1,500 years to see a meaningful improvement of the Caesar cipher in the form of the Vigenère cipher, created in the 16th century by an Italian named Giovan Battista Bellaso. The name *Vigenère* comes from the Frenchman Blaise de Vigenère, who invented a different cipher in the 16th century, but due to historical misattribution, Vigenère's name stuck. Nevertheless, the Vigenère cipher became popular and was later used during the American Civil War by Confederate forces and during WWI by the Swiss Army, among others.

The Vigenère cipher is similar to the Caesar cipher, except that letters aren't shifted by three places but rather by values defined by a *key*, a collection of letters that represent numbers based on their position in the alphabet. For example, if the key is DUH, letters in the plaintext are shifted using the values 3, 20, 7 because *D* is three letters after *A*, *U* is 20 letters after *A*, and *H* is seven letters after *A*. The 3, 20, 7 pattern repeats until you've encrypted the entire plaintext. For example, the word *CRYPTO* would encrypt to *FLFSNV* using DUH as the key: *C* is shifted three positions to *F*,

R is shifted 20 positions to *L*, and so on. Figure 1-3 illustrates this principle when encrypting the sentence *THEY DRINK THE TEA*.

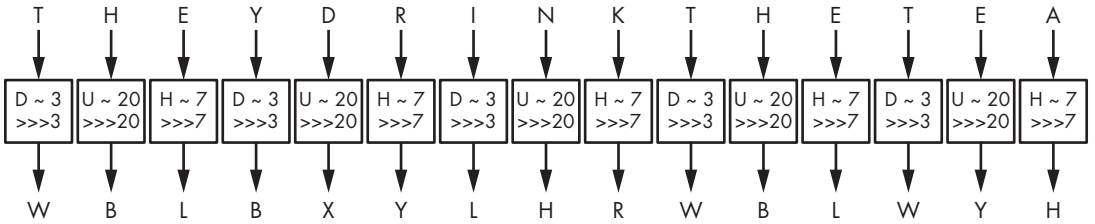


Figure 1-3: The Vigenère cipher

The Vigenère cipher is clearly more secure than the Caesar cipher, yet it's still fairly easy to break. The first step in decrypting it is to figure out the key's length. Take the example in Figure 1-3, wherein *THEY DRINK THE TEA* encrypts to *WBLBXYLHRWBLWYH* with the key DUH. (Spaces are usually removed to hide word boundaries.) Notice that in the ciphertext *WBLBXYLHRWBLWYH*, the group of three letters *WBL* appears twice in the ciphertext at nine-letter intervals. This suggests that the same three-letter word was encrypted using the same shift values, producing *WBL* each time. A cryptanalyst can then deduce that the key's length is either nine or a value that divides nine (that is, three). Furthermore, they may guess that this repeated three-letter word is *THE* and therefore determine DUH as a possible encryption key.

The second step to breaking the Vigenère cipher is to determine the actual key using a method called *frequency analysis*, which exploits the uneven distribution of letters in languages. For example, in English, *E* is the most common letter, so if you find that *X* is the most common letter in a ciphertext, then the most likely plaintext value at this position is *E*.

Despite its relative weakness, the Vigenère cipher may have been good enough to securely encrypt messages in its time. Frequency analysis is limited in that it requires a few sentences, meaning it won't work if the cipher is used to encrypt short messages. Also, most messages needed to be secret for short periods of time, so it didn't matter if ciphertexts were eventually decrypted by the enemy. (The 19th-century cryptographer Auguste Kerckhoffs estimated that most encrypted wartime messages required confidentiality for only three to four hours.)

How Ciphers Work

Basing ourselves on the simplistic Caesar and Vigenère ciphers, we can try to abstract out the workings of a cipher by identifying its two main components: a permutation and a mode of operation. A *permutation* is a function that transforms an item (in cryptography, a letter or a group of bits) such that each item has a unique inverse (for example, the Caesar cipher's

three-letter shift). A *mode of operation* is an algorithm that uses a permutation to process messages of arbitrary size. The mode of the Caesar cipher is trivial—it just repeats the same permutation for each letter—but as you've seen, the Vigenère cipher has a more complex mode, where letters at different positions undergo different permutations.

In the following sections, I discuss in more detail what these components are and how they relate to a cipher's security. I use each component to show why classical ciphers are doomed to be insecure, unlike modern ciphers that run on high-speed computers.

The Permutation

Most classical ciphers work by replacing each letter with another letter—in other words, by performing a *substitution*. In the Caesar and Vigenère ciphers, the substitution is a shift in the alphabet, though the alphabet or set of symbols can vary: instead of the English alphabet, it could be the Arabic alphabet; instead of letters, it could be words, numbers, or ideograms, for example. The representation or encoding of information is a separate matter that is mostly irrelevant to security. (We're considering Latin letters because that's what classical ciphers use.)

A cipher's substitution can't be just any substitution. It should be a permutation, which is a rearrangement of the letters *A* to *Z*, such that each letter has a unique inverse. For example, a substitution that transforms the letters *A*, *B*, *C*, and *D*, respectively, to *C*, *A*, *D*, and *B* is a permutation, because each letter maps to another single letter. But a substitution that transforms *A*, *B*, *C*, *D* to *D*, *A*, *A*, *C* is not a permutation, because both B and C map onto A. With a permutation, each letter has exactly one inverse.

Still, not every permutation is secure. To be secure, a cipher's permutation should satisfy three criteria:

The permutation should be determined by the key This keeps the permutation secret as long as the key is secret. In the Vigenère cipher, if you don't know the key, you don't know which of the 26 permutations was used; hence, you can't easily decrypt.

Different keys should result in different permutations Otherwise, it becomes easier to decrypt without the key: if different keys result in identical permutations, that means there are fewer distinct keys than distinct permutations and therefore fewer possibilities to try when decrypting without the key. In the Vigenère cipher, each letter from the key determines a substitution; there are 26 distinct letters and as many distinct permutations.

The permutation should look random, loosely speaking There should be no pattern in the ciphertext after performing a permutation, because patterns make a permutation predictable for an attacker and therefore less secure. For example, the Vigenère cipher's substitution is pretty predictable: for a given offset, if you determine that *A* encrypts to *F*, you could conclude that the shift value is 5, and you would also know that *B* encrypts to *G*, that *C* encrypts to *H*, and so on. However, with a

randomly chosen permutation, knowing that *A* encrypts to *F* would tell you only that *B* does *not* encrypt to *F*.

We'll call a permutation that satisfies these criteria a *secure permutation*. As you'll see next, a secure permutation is necessary but insufficient on its own for building a secure cipher. A cipher also needs a mode of operation to support messages of any length.

The Mode of Operation

Say we have a secure permutation that transforms *A* to *X*, *B* to *M*, and *N* to *L*, for example. The word *BANANA* therefore encrypts to *MXLXLX*, where each occurrence of *A* is replaced by an *X*. Using the same permutation for all the letters in the plaintext thus reveals any duplicate letters. By analyzing these duplicates, you might not learn the entire message, but you'll learn *something* about it. In the *BANANA* example, you don't need the key to guess that the plaintext's three *X* positions share a letter and that another letter is shared at the two *L* positions. If you know the message is a fruit's name, you could determine that it's *BANANA* rather than *CHERRY*, *LYCHEE*, or another six-letter fruit.

The mode of operation (or *mode*) of a cipher mitigates the exposure of duplicate letters in the plaintext by using different permutations for duplicate letters. The mode of the Vigenère cipher partially addresses this: if the key is *N* letters long, then *N* different permutations will be used for every *N* consecutive letter. However, this can still result in patterns in the ciphertext because every *N*th letter of the message uses the same permutation. That's why frequency analysis works to break the Vigenère cipher.

Frequency analysis can be defeated if the Vigenère cipher encrypts only plaintexts that are of the same length as the key. But even then, there's another problem: reusing the same key several times exposes similarities between plaintexts. For example, with the key KYN, the words *TIE* and *PIE* encrypt to *DGR* and *ZGR*, respectively. Both end with the same two letters (*GR*), revealing that both plaintexts share their last two letters as well. Finding these patterns shouldn't be possible with a secure cipher.

To build a secure cipher, you must combine a secure permutation with a secure mode. Ideally, this combination prevents attackers from learning anything about a message other than its length.

Why Classical Ciphers Are Insecure

Classical ciphers are doomed to be insecure because they're limited to operations you can do in your head or on a piece of paper. They lack the computational power of a computer and are easily broken by simple computer programs. Let's see the fundamental reason why that simplicity makes them insecure in today's world.

Remember that a cipher's permutation should look random to be secure. Of course, the best way to look random is to *be* random—that is, to select every permutation randomly from the set of all permutations. And there are many permutations to choose from. In the case of the 26-letter English alphabet, there are approximately 2^{88} permutations:

$$26! = 403291461126605635584000000 \approx 2^{88}$$

Here, the exclamation point (!) is the factorial symbol, defined as follows:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2$$

(To see why we end up with this number, count the permutations as lists of reordered letters: there are 26 choices for the first possible letter, then 25 possibilities for the second, 24 for the third, and so on.) This number is huge: it's of the same order of magnitude as the number of atoms in the human body. But classical ciphers can use only a small fraction of those permutations—namely, those that require simple operations (such as shifts) and have a short description (like a short algorithm or a small lookup table). The problem is that a secure permutation can't accommodate both of these limitations.

You can get secure permutations using simple operations by picking a random permutation, representing it as a table of 25 letters (enough to represent a permutation of 26 letters, with the 26th one missing), and applying it by looking up letters in this table. But then you wouldn't have a short description. For example, it would take 250 letters to describe 10 different permutations, rather than just the 10 letters used in the Vigenère cipher.

You can also produce secure permutations with a short description. Instead of just shifting the alphabet, you could use more complex operations such as addition and multiplication. This is how modern ciphers work: given a key of typically 128 or 256 bits, they perform hundreds of bit operations to encrypt a single letter. This process is fast on a computer that can do billions of bit operations per second, but it would take hours to do by hand and would still be vulnerable to frequency analysis.

The Perfect Cipher: The One-Time Pad

Essentially, a classical cipher can't be secure unless it comes with a huge key, but encrypting with a huge key is impractical. However, the one-time pad is such a cipher, and it is the most secure cipher. In fact, it guarantees *perfect secrecy*: even if an attacker has unlimited computing power, it's impossible to learn anything about the plaintext except for its length.

In the next sections, I'll show you how a one-time pad works and then offer a sketch of its security proof.

Encryption and Decryption

The one-time pad takes a plaintext, P, and a random key, K, that's the same length as P and produces a ciphertext, C, defined as

$$C = P \oplus K$$

where C, P, and K are bit strings of the same length and $\oplus$ is the bitwise exclusive OR operation (XOR), defined as $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.

NOTE *I'm presenting the one-time pad in its usual form, as working on bits, but it can be adapted to other symbols. With letters, for example, you'd end up with a variant of the Caesar cipher with a shift index picked at random for each letter.*

The one-time pad's decryption is identical to encryption; it's just an XOR: $P = C \oplus K$. Indeed, we can verify $C \oplus K = P \oplus K \oplus K = P$ because XORing K with itself gives the all-zero string 000 . . . 000. That's it—even simpler than the Caesar cipher.

For example, if $P = 01101101$ and $K = 10110100$, then we can calculate the following:

$$C = P \oplus K = 01101101 \oplus 10110100 = 11011001$$

Decryption retrieves P by computing the following:

$$P = C \oplus K = 11011001 \oplus 10110100 = 01101101$$

The important thing is that a one-time pad can be used *one time*: each key K should be used only once. If the same K is used to encrypt P_1 and P_2 to C_1 and C_2, then an eavesdropper can compute the following:

$$C_1 \oplus C_2 = \left(P_1 \oplus K\right) \oplus \left(P_2 \oplus K\right) = P_1 \oplus P_2 \oplus K \oplus K = P_1 \oplus P_2$$

An eavesdropper would thus learn the XOR difference of P_1 and P_2, information that should be kept secret. Moreover, if either plaintext message is known, then the other message can be recovered.

The one-time pad is utterly inconvenient to use because it requires a key as long as the plaintext and a new random key for each new message or group of data. To encrypt a 1TB hard drive, you'd need another 1TB drive to store the key! Nonetheless, the one-time pad has been used throughout history—by the British Special Operations Executive during World War II, by Soviet spies, by the National Security Agency (NSA)—and is still used today in specific contexts. (I've heard of Swiss bankers who couldn't agree on a cipher trusted by both parties and ended up using one-time pads, but I don't recommend doing this.)

Why Is the One-Time Pad Secure?

Although the one-time pad is not practical, it's important to understand what makes it secure. In the 1940s, American mathematician Claude Shannon proved that the one-time pad's key must be at least as long as the message to achieve perfect secrecy. The proof's idea is fairly simple. You assume that the attacker has unlimited power and thus can try all the keys. The goal is to encrypt such that the attacker can't rule out any possible plaintext given some ciphertext.

The intuition behind the one-time pad's perfect secrecy goes as follows: if K is random, the resulting C looks as random as K to an attacker because the XOR of a random string with any fixed string yields a random string. To see

this, consider the probability of getting 0 as the first bit of a random string (a probability of 1/2). What's the probability that a random bit XORed with the second bit is 0? Right, 1/2 again. The same argument can be iterated over bit strings of any length. The ciphertext C thus looks random to an attacker that doesn't know K, so it's literally impossible to learn anything about P given C, even for an attacker with unlimited time and power. In other words, knowing the ciphertext gives no information whatsoever about the plaintext except its length—pretty much the definition of a secure cipher.

For example, if a ciphertext is 128 bits long (meaning the plaintext is 128 bits as well), there are 2^{128} possible ciphertexts; therefore, there should be 2^{128} possible plaintexts from the attacker's point of view. But if there are fewer than 2^{128} possible keys, the attacker can rule out some plaintexts. If the key is only 64 bits, for example, the attacker can determine the 2^{64} possible plaintexts and rule out the overwhelming majority of 128-bit strings. The attacker wouldn't learn what the plaintext is, but they would learn what the plaintext is not, which makes the encryption's secrecy imperfect.

You must have a key as long as the plaintext to achieve perfect security, but this quickly becomes impractical for real-world use. Next, I'll discuss the approaches taken in modern-day encryption to achieve the best security that's both possible and practical.

PROBABILITY IN CRYPTOGRAPHY

A *probability* is a number that expresses the likelihood, or chance, of some event happening. It's expressed as a number between 0 and 1, where 0 means "never" and 1 means "always." The higher the probability, the greater the chance. Many explanations of probability use the example of white balls and red balls in a bag and the probability of picking a ball of either color.

Cryptography often uses probabilities to measure an attack's chances of success, by first counting the number of successful events (for example, the event "find the correct secret key") and then counting the total number of possible events (for example, the total number of keys is 2^n if we deal with n-bit keys). In this example, the probability that a randomly chosen key is the correct one is $1/2^n$, or the count of successful events (1 secret key) and the count of possible events (2^n possible keys). The number $1/2^n$ is negligibly small for common key lengths such as 128 and 256.

The probability of an event *not happening* is $1 - p$, where p is the event's probability. The probability of getting a wrong key in our example is therefore $1 - 1/2^n$, a number very close to 1, meaning near certainty.

Encryption Security

Classical ciphers aren't secure, but a perfectly secure cipher like the one-time pad is impractical. We'll thus have to give a little in terms of security if we want secure *and* usable ciphers. But what does *secure* really mean, besides the obvious and informal "eavesdroppers can't decrypt secure messages"?

A cipher is secure if, even given a large number of plaintext–ciphertext pairs, *nothing can be learned* about the cipher's behavior when applied to other plaintexts or ciphertexts. This opens up new questions:

- How does an attacker come by these pairs? How large is a "large number"? This is all defined by *attack models*, assumptions about what the attacker can and cannot do.
- What could be "learned," and what "cipher's behavior" are we talking about? This is defined by *security goals*, descriptions of what is considered a successful attack.

Attack models and security goals must go together; you can't claim that a system is secure without explaining against whom or from what it's safe. A *security notion* is the combination of a security goal with an attack model. We'll say that a cipher *achieves* a certain security notion if any attacker working in a given model can't break the security goal.

Attack Models

An attack model is a set of assumptions about how attackers might interact with a cipher and what they can and can't do. The goals of an attack model are as follows:

- To set requirements for cryptographers who design ciphers so that they know what attackers and what kinds of attacks to protect against.
- To give guidelines to users about whether a cipher will be safe to use in their environment.
- To provide clues for cryptanalysts who attempt to break ciphers so they know whether a given attack is valid. An attack is valid only if it's doable in the model considered.

Attack models don't need to match reality exactly; they're an approximation. As the statistician George E. P. Box put it, "All models are wrong; the practical question is how wrong do they have to be to not be useful." To be useful in cryptography, attack models should at least encompass what attackers can actually do to attack a cipher. It's beneficial if a model overestimates attackers' capabilities because it helps anticipate future attack techniques—only the paranoid cryptographers survive. A bad model underestimates attackers and provides false confidence in a cipher by making it seem secure in theory when it's not secure in reality.

Kerckhoffs's Principle

One assumption made in all models is *Kerckhoffs's principle*, which states that the security of a cipher should rely only on the secrecy of the key and not on the secrecy of the cipher. This may sound obvious today, when ciphers and protocols are publicly specified and used by everyone. But historically, Dutch linguist Auguste Kerckhoffs was referring to military encryption machines specifically designed for a given army or division. Quoting from his 1883 essay, "La Cryptographie Militaire," where he listed six requirements of a military encryption system: "The system must not require secrecy and can be stolen by the enemy without causing trouble."

Black-Box Models

Let's consider some useful attack models expressed in terms of what the attacker can observe and what queries they can make to the cipher. A *query* for our purposes is the operation that sends an input value to some function and gets the output in return, without exposing the details of that function. An *encryption query*, for example, takes a plaintext and returns a corresponding ciphertext, without revealing the secret key.

We call these *black-box models* because the attacker sees only what goes in and out of the cipher. For example, some smart card chips securely protect a cipher's internals as well as its keys, yet you're allowed to connect to the chip and ask it to decrypt any ciphertext. The attacker would then receive the corresponding plaintext, which may help them determine the key. That's a real example where *decryption queries* are possible.

There are several different black-box attack models. Here, I list them in order from weakest to strongest, describing attackers' capabilities for each model:

Ciphertext-only attackers (COAs) Observe ciphertexts but don't know the associated plaintexts or how the plaintexts were selected. Attackers in the COA model are passive and can't perform encryption or decryption queries.

Known-plaintext attackers (KPAs) Observe ciphertexts and do know the associated plaintexts. Attackers in the KPA model thus get a list of plaintext–ciphertext pairs, where plaintexts are assumed to be randomly selected. KPA is a passive attacker model.

Chosen-plaintext attackers (CPAs) Can perform encryption queries for plaintexts of their choice and observe the resulting ciphertexts. This model captures situations where attackers can choose all or part of the encrypted plaintexts and then get to see the ciphertexts. Unlike COA or KPA, which are passive models, CPAs are *active* attackers because they influence the encryption processes rather than passively eavesdropping.

Chosen-ciphertext attackers (CCAs) Can both encrypt and decrypt; that is, they get to perform encryption queries and decryption queries (of ciphertexts different from the targeted ciphertext). The CCA model

may sound ludicrous at first—if you can decrypt, what else do you need?—but like the CPA model, it aims to represent situations where attackers can have some influence on the ciphertext and later get access to the plaintext. Moreover, decryption is not always enough to break a system. For example, some video-protection devices allow attackers to perform encryption queries and decryption queries using the device's chip, but in that context, attackers are interested in the key in order to redistribute it; in this case, being able to decrypt "for free" isn't sufficient to break the system.

In the preceding models, ciphertexts that are observed as well as queried don't come for free. Each ciphertext comes from the computation of the encryption function. This means that generating 2^N plaintext–ciphertext pairs through encryption queries takes about as much computation as trying 2^N keys, for example. The cost of queries should be taken into account when computing the cost of an attack.

Gray-Box Models

In a *gray-box model*, the attacker has access to a cipher's *implementation*. This makes gray-box models more realistic than black-box models for applications such as smart cards, embedded systems, and virtualized systems, to which attackers often have physical access and can thus tamper with the algorithms' internals. By the same token, gray-box models are more difficult to define than black-box ones because they depend on physical, analog properties rather than just on an algorithm's input and outputs, and crypto theory often fails to abstract the complexity of the real world.

Side-channel attacks are a family of attacks within gray-box models. A side channel is a source of information that depends on the implementation of the cipher, be it in software or in hardware. Side-channel attackers observe or measure analog characteristics of a cipher's implementation but don't alter its integrity; they are *noninvasive*. For pure software implementations, typical side channels are the execution time and the behavior of the system that surrounds the cipher, such as error messages, return values, and branches. In the case of implementations on smart cards, for example, typical side-channel attackers measure power consumption, electromagnetic emanations, or acoustic noise.

Invasive attacks are a family of attacks on cipher implementations that are more powerful than side-channel attacks and are more expensive because they often require sophisticated equipment. You can run basic side-channel attacks with a standard PC and an off-the-shelf oscilloscope, but invasive attacks may require tools such as a high-resolution microscope and a chemical lab. Invasive attacks consist of a whole set of techniques and procedures, including using nitric acid to remove a chip's packaging, acquiring microscopic imagery, partial reverse engineering, and modifying the chip's behavior with techniques such as laser fault injection and electromagnetic injections.

Security Goals

I've informally defined the goal of security as "nothing can be learned about the cipher's behavior." To turn this idea into a rigorous mathematical definition, cryptographers define two security goals that correspond to different ideas of what it means to learn about a cipher's behavior:

Indistinguishability (IND) Ciphertexts should be indistinguishable from random strings. This is usually illustrated with a hypothetical game: if an attacker picks two plaintexts and then receives a ciphertext of one of the two (chosen at random), they shouldn't be able to tell which plaintext was encrypted, even by performing encryption queries with the two plaintexts (and decryption queries, if the model is CCA rather than CPA).

Nonmalleability (NM) Given a ciphertext $C_1 = \mathbf{E}(K, P_1)$, it should be impossible to create another ciphertext, C_2, whose corresponding plaintext, P_2, is related to P_1 in a meaningful way (for example, to create a P_2 that is equal to $P_1 \oplus 1$ or to $P_1 \oplus X$ for some known value, X). Surprisingly, the one-time pad is malleable: given a ciphertext $C_1 = P_1 \oplus K$, you can define $C_2 = C_1 \oplus 1$, which is a valid ciphertext of $P_2 = P_1 \oplus 1$ under the same key K. Oops, so much for our perfect cipher.

Next, I'll discuss these security goals in the context of different attack models.

Security Notions

Security goals are useful only when combined with an attack model. The convention is to write a security notion as *GOAL-MODEL*. For example, IND-CPA denotes indistinguishability against chosen-plaintext attackers, NM-CCA denotes nonmalleability against chosen-ciphertext attackers, and so on. Let's start with the security goals for an attacker.

Semantic Security and Randomized Encryption: IND-CPA

The most important security notion is IND-CPA, also called *semantic security*. It captures the intuition that ciphertexts shouldn't leak any information about plaintexts as long as the key is secret. To achieve IND-CPA security, encryption must return different ciphertexts if called twice on the same plaintext; otherwise, an attacker could identify duplicate plaintexts from their ciphertexts, contradicting the definition that ciphertexts shouldn't reveal any information. But note that even the IND-CPA-secure scheme will inevitably leak one piece of information about the plaintext: its length, or at least approximate length. This is why encrypting compressed data is generally not a good idea, as the size of the compressed data can reveal information on the original data.

One way to achieve IND-CPA security is to use *randomized encryption*. As the name suggests, it randomizes the encryption process and returns different ciphertexts when the same plaintext is encrypted twice. Encryption

can then be expressed as $C = \mathbf{E}(K, R, P)$, where R is fresh random bits. Decryption remains deterministic, however, because given $\mathbf{D}(K, R, C)$, you should always get P, regardless of the value of R.

What if encryption isn't randomized? In the IND game introduced in the "Security Goals" section on the previous page, the attacker picks two plaintexts, P_1 and P_2, and receives a ciphertext of one of the two but doesn't know which plaintext the ciphertext corresponds to. That is, they get $C_i = \mathbf{E}(K, P_i)$ and have to guess whether i is 1 or 2. In the CPA model, the attacker can perform encryption queries to determine both $C_1 = \mathbf{E}(K, P_1)$ and $C_2 = \mathbf{E}(K, P_2)$. If encryption isn't randomized, it suffices to see if C_i is equal to C_1 or to C_2 in order to determine which plaintext was encrypted and thereby win the IND game. Therefore, randomization is key to the IND-CPA notion.

If you don't have a pseudorandom generator, you may still achieve IND-CPA security by using an encryption scheme that requires a *nonce* (or *number used only once*), rather than a random, unpredictable value. A nonce must be unique for every new encryption call. A mere counter (1, 2, 3, . . .) would do the trick. For example, AES-CTR (the AES block cipher used in CTR mode) is IND-CPA if its additional input, the nonce, is unique. Unlike some randomized encryption schemes that just use randomness as part of the encryption process, the nonce is *necessary to decrypt* with algorithms using nonces.

NOTE *With randomized encryption, ciphertexts must be slightly longer than plaintexts to allow for more than one possible ciphertext per plaintext. For example, if there are 2^{64} possible ciphertexts per plaintext, ciphertexts must be at least 64 bits longer than plaintexts.*

Semantically Secure Encryption

One of the simplest constructions of a semantically secure cipher uses a *deterministic random bit generator (DRBG)*, an algorithm that returns random-looking bits given some secret value:

$$\mathbf{E}(K, R, P) = \left(\mathbf{DRBG}(K \,||\, R) \oplus P, R\right)$$

Here, R is a string randomly chosen for each new encryption and given to a DRBG along with the key ($K \,||\, R$ denotes the string consisting of K followed by R). This approach is reminiscent of the one-time pad: instead of picking a random key of the same length as the message, we leverage a random bit generator to get a random-looking string.

The proof that this cipher is IND-CPA secure is simple, if we assume that the DRBG produces random bits. The proof works ad absurdum: if you can distinguish ciphertexts from random strings, which means you can distinguish $\mathbf{DRBG}(K \,||\, R) \oplus P$ from random, then this means you can distinguish $\mathbf{DRBG}(K \,||\, R)$ from random. Remember that the CPA model lets you get ciphertexts for chosen values of P, so you can XOR P to $\mathbf{DRBG}(K \,||\, R) \oplus P$ and get $\mathbf{DRBG}(K \,||\, R)$. But now we have a contradiction because we started

by assuming that **DRBG**($K \| R$) can't be distinguished from random, producing random strings. So we conclude that ciphertexts can't be distinguished from random strings and therefore that the cipher is secure.

NOTE *As an exercise, try to determine what other security notions are satisfied by the cipher* ***E***(K, R, P) = (***DRBG***(K || R) ⊕ P, R)*. Is it NM-CPA? IND-CCA? You'll find the answers in the next section.*

Security Notions Comparisons

You've learned that attack models such as CPA and CCA are combined with security goals such as NM and IND to build the security notions NM-CPA, NM-CCA, IND-CPA, and IND-CCA. How are these notions related? Can we prove that satisfying notion X implies satisfying notion Y?

Some relations are obvious: IND-CCA implies IND-CPA, and NM-CCA implies NM-CPA because anything a CPA attacker can do, a CCA attacker can do as well. That is, if you can't break a cipher by performing chosen-ciphertext and chosen-plaintext queries, you can't break it by performing only chosen-plaintext queries.

A less obvious relation is that IND-CPA does not imply NM-CPA. To understand this, observe that the previous IND-CPA construction (**DRBG**(K, R) ⊕ P, R) is not NM-CPA: given a ciphertext (X, R), you can create the ciphertext (X ⊕ 1, R), which is a valid ciphertext of P ⊕ 1, thus contradicting the notion of nonmalleability.

But the opposite relation holds: NM-CPA implies IND-CPA. The intuition is that IND-CPA encryption is like putting items in a bag: you don't get to see them, but you can rearrange their positions in the bag by shaking it up and down. NM-CPA is more like a safe: once inside, you can't interact with what you put in there. This analogy doesn't work for IND-CCA and NM-CCA, which are equivalent notions that each imply the presence of the other. I'll spare you the proof, which is pretty technical.

TWO TYPES OF ENCRYPTION APPLICATIONS

There are two main types of encryption applications. *In-transit encryption* protects data sent from one machine to another: data is encrypted before being sent and decrypted after being received, as in encrypted connections to e-commerce websites. *At-rest encryption* protects data stored on an information system. Data is encrypted before being written to memory and decrypted before being read. Examples include disk encryption systems on laptops as well as virtual machine encryption for cloud virtual instances. The security notions we've seen apply to both types of applications, but the right notion to consider may depend on the application.

Asymmetric Encryption

So far, we've considered only symmetric encryption, where two parties share a key. In *asymmetric encryption*, there are two keys: one encrypts and the other decrypts. The encryption key is called a *public key* and is generally considered publicly available to anyone who wants to send you encrypted messages. The decryption key, however, must remain secret and is called a *private key*.

The public key can be computed from the private key, but the private key can't be computed from the public key. In other words, it's easy to compute in one direction but not in the other—and that's the point of *public-key cryptography*, whose functions are easy to compute in one direction but practically impossible to invert.

The attack models and security goals for asymmetric encryption are about the same as for symmetric encryption, except that because the encryption key is public, any attacker can make encryption queries by using the public key to encrypt. The default model for asymmetric encryption is therefore the chosen-plaintext attacker.

Symmetric and asymmetric encryption are the two main types of encryption, and they are usually combined to build secure communication systems. They also form the basis of more sophisticated schemes, as you'll see next.

When Ciphers Do More Than Encryption

Basic encryption turns plaintexts into ciphertexts and ciphertexts into plaintexts, with no requirements other than security. However, some applications often need more than that, be it extra security features or functionalities. That's why cryptographers created variants of symmetric and asymmetric encryption. Some are well understood, efficient, and widely deployed, while others are experimental, hardly used, and offer poor performance.

Authenticated Encryption

Authenticated encryption (AE) is a type of symmetric encryption that returns an *authentication tag* in addition to a ciphertext. Figure 1-4 shows authenticated encryption sets $\mathbf{AE}(K, P) = (C, T)$, where the authentication tag T is a short string that's impossible to guess without the key. Decryption takes K, C, and T and returns the plaintext P only if it verifies that T is a valid tag for that plaintext–ciphertext pair; otherwise, it aborts and returns some error.

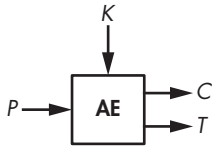


Figure 1-4: Authenticated encryption

The tag ensures the *integrity* of the message and serves as evidence that the ciphertext received is identical to the one sent in the first place by a legitimate party that knows the key *K*. When *K* is shared with only one other party, the tag also guarantees that the message was sent by that party; that is, it implicitly *authenticates* the expected sender as the actual creator of the message.

NOTE *I use "creator" rather than "sender" here because an eavesdropper can record some (*C, T*) pairs sent by party A to party B and then send them again to B, pretending to be A. This is called a* replay attack*, and it can be prevented—for example, by including a counter number in the message. When a message is decrypted, its counter* i *increases by one:* i + *1. In this way, one could check the counter to see if a message has been sent twice, indicating that an attacker is attempting a replay attack by resending the message. This also enables the detection of lost messages.*

Authenticated encryption with associated data (AEAD) is an extension of authenticated encryption that takes some cleartext and unencrypted data and uses it to generate the authentication tag **AEAD**(*K*, *P*, *A*) = (*C*, *A*, *T*). A typical application of AEAD protects protocols' datagrams with a cleartext header and an encrypted payload. In such cases, at least some header data has to remain in the clear; for example, destination addresses need to be clear to route network packets.

For more on authenticated encryption, jump to Chapter 8.

Format-Preserving Encryption

A basic cipher takes bits and returns bits; it doesn't care whether bits represent text, an image, or a PDF document. The ciphertext may in turn be encoded as raw bytes, hexadecimal characters, base64, and other formats. But what if you need the ciphertext to have the same format as the plaintext, as is sometimes required by database systems that can record data only in a prescribed format?

Format-preserving encryption (FPE) solves this problem. It can create ciphertexts that have the same format as the plaintext. For example, FPE can encrypt IP addresses to IP addresses (as shown in Figure 1-5), ZIP codes to ZIP codes, credit card numbers to credit card numbers with a valid checksum, and so on.

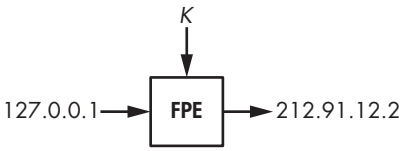


Figure 1-5: Format-preserving encryption for IP addresses

Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) is the holy grail to cryptographers: it enables its users to replace a ciphertext, $C = \mathbf{E}(K, P)$, with another ciphertext, $C' = \mathbf{E}(K, \mathbf{F}(P))$, where $\mathbf{F}(P)$ can be any function of P, without ever decrypting the initial ciphertext, C. For example, P can be a text document and **F** can be the modification of part of the text. Imagine a cloud application that stores your encrypted data, but the cloud provider doesn't know what the data is or the type of changes made when you modify that data. Sounds amazing, doesn't it?

But there's a flip side: this type of encryption is slow—so slow that even the most basic operation would take an unacceptably long time. The first FHE scheme was created in 2009, and since then more efficient variants have appeared, but it remains unclear whether FHE will ever be fast enough to be useful. However, application-specific use cases of (partially) homomorphic encryption have proved more efficient for operations such as evaluating machine learning models.

Searchable Encryption

Searchable encryption enables searching over an encrypted database without leaking the searched terms by encrypting the search query itself. Like fully homomorphic encryption, searchable encryption could enhance the privacy of many cloud-based applications by hiding your searches from your cloud provider. Some commercial solutions claim to offer searchable encryption, though they're mostly based on standard cryptography with a few tricks to enable partial searchability. As of this writing, however, searchable encryption remains experimental within the research community.

Tweakable Encryption

Tweakable encryption (TE) is similar to basic encryption, except for an additional parameter called the *tweak*, which aims to simulate different versions of a cipher (see Figure 1-6). The tweak might be a unique per-customer value to ensure that a customer's cipher can't be cloned by other parties using the same product, but the main application of TE is *disk encryption*. However, TE is not bound to a single application and is a lower-level type of encryption used to build other schemes, such as authentication encryption modes.

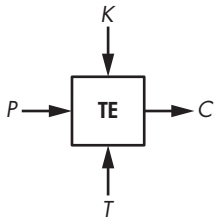


Figure 1-6: Tweakable encryption

In disk encryption, TE encrypts the content of storage devices such as hard drives or solid-state drives. (Randomized encryption can't be used because it increases the size of the data, which is unacceptable for files on storage media.) To make encryption unpredictable, TE uses a tweak value that depends on the position of the data encrypted, which is usually a sector number or a block index.

How Things Can Go Wrong

Encryption algorithms or implementations thereof can fail to protect confidentiality in many ways. This can be due to a failure to match the security requirements (such as "be IND-CPA secure") or to set requirements matching reality (if you target only IND-CPA security when attackers can actually perform chosen-ciphertext queries). Alas, many engineers don't even think about cryptographic security requirements and want to be "secure" without understanding what that actually means. That's usually a recipe for disaster. Let's look at two examples.

Weak Cipher

Our first example concerns ciphers that can be attacked using cryptanalysis techniques, as occurred with the 2G mobile communication standard. Encryption in 2G mobile phones used a cipher called A5/1 that turned out to be weaker than expected, enabling the interception of calls by anyone with the right skills and tools. Telecommunication operators had to find workarounds to prevent the attack.

NOTE *The 2G standard also defined A5/2, a cipher for areas other than the European Union and United States. A5/2 was purposefully weaker to prevent the use of strong encryption everywhere.*

That said, attacking A5/1 isn't trivial, and it took more than 10 years for researchers to come up with an effective cryptanalysis method. Furthermore, the attack is a *time-memory trade-off (TMTO)*, a type of method that first runs computations for days or weeks to build large lookup tables, which are subsequently used for the actual attack. For A5/1, the precomputed tables are

of the order of 1TB. Later standards for mobile encryption, such as 3G and LTE, specify stronger ciphers, but that doesn't mean their encryption won't be compromised; it simply means that the encryption won't be compromised by breaking the symmetric cipher that's part of the system.

Wrong Model

The next example concerns an invalid attack model that overlooked some side channels.

Many communication protocols that use encryption ensure that they use ciphers considered secure in the CPA or CCA model. However, some attacks don't require encryption queries, as in the CPA model, nor do they require decryption queries, as in the CCA model. They simply need *validity queries* to tell whether a ciphertext is valid, and these queries are usually sent to the system responsible for decrypting ciphertexts. *Padding oracle attacks* are an example of such attacks, wherein an attacker learns whether a ciphertext conforms to the required format.

Specifically, in the case of padding oracle attacks, a ciphertext is valid only if its plaintext has the proper *padding*, a sequence of bytes appended to the plaintext to simplify encryption. Decryption fails if the padding is incorrect, and attackers can often detect decryption failures and attempt to exploit them. For example, the presence of the Java exception `javax.crypto.BadPaddingException` indicates that an incorrect padding was observed.

In 2010, researchers found padding oracle attacks in several web application servers. The validity queries consisted of sending a ciphertext to some system and observing whether it threw an error. Thanks to these queries, they could decrypt otherwise-secure ciphertexts without knowing the key.

Cryptographers often overlook attacks like padding oracle attacks because they usually depend on an application's behavior and on how users can interact with the application. But if you don't anticipate such attacks and fail to include them in your model when designing and deploying cryptography, you may have some nasty surprises.

Further Reading

We discuss encryption and its various forms in more detail throughout this book, especially how modern, secure ciphers work. Still, I can't cover everything and have passed over many fascinating topics. For example, to learn the theoretical foundations of encryption and gain a deeper understanding of the notion of indistinguishability, read the 1982 paper that introduced the idea of semantic security, "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information" by Goldwasser and Micali. If you're interested in physical attacks and cryptographic hardware, the proceedings of the Cryptographic Hardware and Embedded Systems (CHES) conference are the main reference.

There are also many more types of encryption than those presented in this chapter, including attribute-based encryption, broadcast encryption, functional encryption, identity-based encryption, message-locked encryption, and proxy reencryption, to cite but a few. For the latest research on those topics, check *https://eprint.iacr.org*, an electronic archive of cryptography research papers.

2

RANDOMNESS

Randomness is found everywhere in cryptography: in the generation of secret keys, in encryption schemes, and even in the attacks on cryptosystems. Without randomness, cryptography would be impossible because all operations would become predictable and therefore insecure.

This chapter introduces the concept of randomness in the context of cryptography and its applications. We discuss pseudorandom number generators and how operating systems can produce reliable randomness, and we conclude with real examples showing how flawed randomness can impact security.

Random or Nonrandom?

You've probably heard the phrase *random bits* before, but strictly speaking, there is no such thing as a series of random bits. What *is* random is the

algorithm, or process, that produces a series of random bits; therefore, when we say "random bits," we actually mean randomly generated bits.

What do random bits look like? For example, the 8-bit string 11010110 might look more random than 00000000, although both have the same chance of being generated (namely, 1/256). The value 11010110 looks more random than 00000000 because it has the signs typical of a randomly generated value. That is, 11010110 has no obvious pattern.

When we see the string 11010110, our brain registers that it has 3 zeros and 5 ones, just like 55 other 8-bit strings (11111000, 11110100, 11110010, and so on), but only one 8-bit string has 8 zeros. Because the pattern 3-zeros-and-5-ones is more likely to occur than the pattern 8-zeros, we identify 11010110 as random and 00000000 as nonrandom, even if they're not.

This example illustrates two types of errors people often make when identifying randomness:

Mistaking nonrandomness for randomness Thinking that an object was randomly generated simply because it *looks* random

Mistaking randomness for nonrandomness Thinking that patterns appearing by chance are there for a reason other than chance

The distinction between random-looking and actually random is crucial. Indeed, in crypto, nonrandomness is often synonymous with insecurity.

The saying "it happened by chance" reflects the property that from a complex system (in this case, our universe that obeys the laws of physics, deterministic at the macroscopic level and truly random at the subatomic, quantum level) can emerge specific patterns, such as the string 00000000. By the law of large numbers, if many events occur, some won't look random—such as a series of sequential numbers in a lottery draw. Many pseudosciences and belief systems are in fact cases of mistaking randomness for nonrandomness.

Randomness as a Probability Distribution

Any randomized process is characterized by a *probability distribution*, which gives all there is to know about the randomness of the process. A probability distribution, or simply *distribution*, lists the outcomes of a randomized process where each outcome is assigned a probability.

A *probability* measures the likelihood of an event occurring. It's expressed as a real number between 0 and 1 where a probability of 0 means impossible and a probability of 1 means certain. For example, when tossing a two-sided coin, each side has a 1/2 (or 0.5) probability of landing face up, and the probability of a coin landing on its edge has a probability close to 0.

A probability distribution must include all possible outcomes such that the sum of all probabilities is 1. Specifically, if there are N possible events, there are N probabilities $p_1, p_2, \ldots, p_N$ with $p_1 + p_2 + \ldots + p_N = 1$. In the case of the coin toss, the distribution is 1/2 for heads and 1/2 for tails. The sum

of both probabilities is equal to 1/2 + 1/2 = 1, because the coin will fall on one of its two faces.

A *uniform distribution* occurs when all probabilities in the distribution are equal, meaning that all outcomes are equally likely to occur. If there are N events, then each event has probability $1/N$. For example, if a 128-bit key is picked uniformly at random—that is, according to a uniform distribution—then each of the 2^{128} possible keys should have a probability of $1/2^{128}$.

In contrast, when a distribution is *nonuniform*, probabilities aren't all equal. A coin toss with a nonuniform distribution is said to be biased and may yield heads with probability 1/4 and tails with probability 3/4, for example.

NOTE *It's possible to cheat with a loaded die, preventing the probabilities of each of the six faces to be 1/6; however, one can't bias a coin. Coin tosses can be biased only if "the coin is allowed to bounce or be spun rather than simply flipped in the air," as described in the article "You Can Load a Die but You Can't Bias a Coin" (available at* https://www.stat.berkeley.edu/~nolan/Papers/dice.pdf*).*

Entropy: A Measure of Uncertainty

Entropy is the measure of uncertainty, or disorder, in a system. The higher the entropy, the less certainty found in the result of a randomized process.

We can compute the entropy of a probability distribution. If your distribution consists of probabilities p_1, p_2, . . . , p_N, then its entropy is the negative sum of all probabilities multiplied by their logarithm, as shown in this expression:

$$-p_1 \times \log(p_1) - p_2 \times \log(p_2) - \ldots\ p_N \times \log(p_N)$$

Here the function *log* is the *binary logarithm*, or logarithm in base two. Unlike the natural logarithm, the binary logarithm expresses the information in bits and yields integer values when probabilities are powers of two. For example, $\log(1/2) = -1$, $\log(1/4) = -2$, and more generally $\log(1/2^n) = -n$. (We actually take the *negative sum* to end up with a positive number.) Random 128-bit keys produced using a uniform distribution therefore have the following entropy:

$$2^{128} \times \left(-2^{-128} \times \log\left(2^{-128}\right)\right) = -\log\left(2^{-128}\right) = 128 \text{ bits}$$

If you replace 128 with any integer n, the entropy of a uniformly distributed n-bit string will be n bits.

Entropy is maximized when the distribution is uniform because a uniform distribution maximizes uncertainty: no outcome is more likely than the others. Therefore, n-bit values can't have more than n bits of entropy.

By the same token, when the distribution is not uniform, entropy is lower. Consider the coin toss example. The entropy of a fair toss is the following:

$$-(1/2) \times \log(1/2) - (1/2) \times \log(1/2) = 1/2 + 1/2 = 1 \text{ bit}$$

What if one side of the coin has a higher probability of landing face up than the other? Say heads has a probability of 1/4 and tails 3/4. (Remember that the sum of all probabilities should be 1.)

The entropy of such a biased toss is this:

$$-(3/4) \times \log(3/4) - (1/4) \times \log(1/4) \approx -(3/4) \times (-0.415) - (1/4) \times (-2) \approx 0.81 \text{ bits}$$

The fact that 0.81 is less than the 1-bit entropy of a fair toss tells us that the more biased the coin, the less uniform the distribution and the lower the entropy. Taking this example further, if heads has a probability of 1/10, the entropy is 0.469; if the probability drops to 1/100, the entropy drops to 0.081.

NOTE *Entropy can also be viewed as a measure of information. For example, the result of a fair coin toss gives you exactly 1 bit of information—heads or tails—and you're unable to predict the result of the toss in advance. In the case of the unfair coin toss, you know in advance that tails is more probable, so you can predict the outcome. The result of the unfair coin toss gives you the information needed to predict the result with certainty.*

Random and Pseudorandom Number Generators

Cryptosystems need randomness to be secure and therefore need a component from which to get their randomness. The job of this component is to return random bits when requested to do so. To perform this randomness generation, you'll need two things:

- A source of entropy, provided by random number generators.
- A cryptographic algorithm to produce high-quality random bits from the source of entropy. This is found in pseudorandom number generators.

Using both random and pseudorandom number generators is the key to making cryptography practical and secure. Let's briefly look at how random number generators work before exploring pseudorandom number generators in depth.

Randomness comes from the environment, which is analog, chaotic, uncertain, and hence unpredictable. Randomness can't be generated by computer-based algorithms alone. In cryptography, randomness usually comes from *random number generators (RNGs)*, which are software or hardware components that leverage entropy in the analog world to produce unpredictable bits in a digital system. For example, an RNG might directly

sample bits from measurements of temperature, acoustic noise, air turbulence, or electrical static. Unfortunately, such analog entropy sources aren't always available, and their entropy is often difficult to estimate.

RNGs can also harvest the entropy in a running operating system by drawing from attached sensors, I/O devices, network or disk activity, system logs, running processes, and user activities such as key presses and mouse movement. Such system- and human-generated activities can be a good source of entropy, but they can be fragile and manipulated by an attacker. Also, they're slow to yield random bits.

NOTE Quantum random number generators (QRNGs) *are a type of RNG that rely on the randomness arising from quantum mechanical phenomena, such as radioactive decay, photon polarization, or thermal noise. These phenomena, not being characterized by equations that determine the future state from the current state, are random in the absolute sense. In practice, however, the raw bits extracted from a QRNG may be biased and tend to be slow to produce. Like the previously cited entropy sources, they require postprocessing to generate reliable bits at high speed.*

Pseudorandom number generators (PRNGs) address the challenge in generating randomness by reliably producing many artificial random bits from a few true random bits. For example, an RNG that translates mouse movements to random bits would stop working if you stop moving the mouse, whereas a PRNG always returns pseudorandom bits when requested to do so.

PRNGs rely on RNGs but behave differently: RNGs produce true random bits relatively slowly from analog sources, in a nondeterministic way, and with no guarantee of uniform distribution or of high entropy per bit. In contrast, PRNGs produce random-looking bits quickly from digital sources, in a deterministic way, uniformly distributed, and with an entropy guaranteed to be high enough for cryptographic applications. Essentially, PRNGs transform a few unreliable random bits into a long stream of reliable pseudorandom bits suitable for crypto applications, as Figure 2-1 shows.

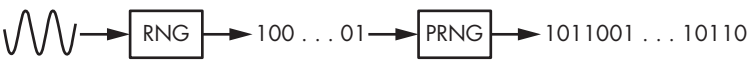


Figure 2-1: RNGs produce few unreliable bits from analog sources, whereas PRNGs expand those bits to a long stream of reliable bits.

How PRNGs Work

A PRNG receives random bits from an RNG at regular intervals and uses them to update the contents of a large memory buffer, called the *entropy pool*. The entropy pool is the PRNG's source of entropy, just like the physical environment is to an RNG. When the PRNG updates the entropy pool, it mixes the pool's bits together to help remove any statistical bias.

To generate pseudorandom bits, the PRNG runs a deterministic random bit generator (DRBG) algorithm that expands some bits from the entropy pool into a much longer sequence. As its name suggests, a DRBG is deterministic, not randomized: given one input, you will always get the same output. The PRNG ensures that its DRBG never receives the same input twice so it can generate unique pseudorandom sequences.

In the course of its work, the PRNG performs three operations:

***init*()** Initializes the entropy pool and the internal state of the PRNG

***refresh*(*R*)** Updates the entropy pool using some data, *R*, usually sourced from an RNG

***next*(*N*)** Returns *N* pseudorandom bits and updates the entropy pool

The *init* operation resets the PRNG to a fresh state, reinitializes the entropy pool to some default value, and initializes any variables or memory buffers used by the PRNG to carry out the *refresh* and *next* operations.

The *refresh* operation is often called *reseeding*, and its argument *R* is called a *seed*. When no RNG is available, seeds may be unique values hard-coded in a system. The *refresh* operation is typically called by the operating system, whereas *next* is typically called or requested by applications. The *next* operation runs the DRBG and modifies the entropy pool to ensure that the next call will yield different pseudorandom bits.

Security Concerns

Let's talk briefly about how PRNGs address high-level security concerns. Specifically, PRNGs should guarantee *backtracking resistance* and *prediction resistance*. Backtracking resistance (also called *forward secrecy*) means that previously generated bits are impossible to recover, whereas prediction resistance (*backward secrecy*) means that future bits should be impossible to predict.

To achieve backtracking resistance, the PRNG should ensure that the transformations performed when updating the state through the *refresh* and *next* operations are irreversible. This way, if an attacker compromises the system and obtains the entropy pool's value, they can't determine the previous values of the pool or the previously generated bits. To achieve prediction resistance, the PRNG should call *refresh* regularly with *R* values that are unknown to an attacker and are difficult to guess, thus preventing an attacker from determining future values of the entropy pool, even if the whole pool is compromised. (If the list of *R* values were known, you'd need to know the order in which *refresh* and *next* calls were made to reconstruct the pool.)

The PRNG Fortuna

Fortuna is a PRNG construction used in Windows originally designed in 2003 by Niels Ferguson and Bruce Schneier. Fortuna superseded *Yarrow*, a 1998 design by John Kelsey and Bruce Schneier that was for a long time used in the macOS and iOS operating systems and has been replaced by

Fortuna. I won't provide the Fortuna specification here or show you how to implement it, but I will try to explain how it works. You'll find a complete description of Fortuna in Chapter 9 of *Cryptography Engineering* by Ferguson, Schneier, and Kohno (Wiley, 2010).

Fortuna's internal memory includes the following:

- Thirty-two entropy pools, P_1, P_2, . . . , P_{32}, such that P_i is used every 2^i reseeds.
- A key, K, and a counter, C (both 16 bytes). These form the internal state of Fortuna's DRBG.

In simplest terms, Fortuna works like this:

- *init*() sets K and C to zero and empties the 32 entropy pools P_i, where $i = 1 \ldots 32$.
- *refresh*(R) appends the data, R, to one of the entropy pools. The system chooses the RNGs used to produce R values, and it should call *refresh* regularly.
- *next*(N) updates K using data from one or more entropy pools, where the choice of the entropy pools depends mainly on how many updates of K have already been done. The N bits requested are then produced by encrypting C using K as a key. If encrypting C is not enough, Fortuna encrypts $C + 1$, then $C + 2$, and so on, to get enough bits.

Although Fortuna's operations look fairly simple, implementing them correctly is hard. For one, you need to get all the details of the algorithm right—how entropy pools are chosen, the type of cipher to be used in *next*, how to behave when no entropy is received, and so on. Although the specs define most of the details, they don't include a comprehensive test suite to check that an implementation is correct, which makes it difficult to ensure that your implementation of Fortuna will behave as expected.

Even if Fortuna is correctly implemented, security failures may occur for reasons other than the use of an incorrect algorithm. For example, Fortuna might not notice if the RNGs fail to produce enough random bits, and as a result Fortuna will produce lower-quality pseudorandom bits, or it may stop delivering pseudorandom bits altogether.

Another risk inherent in Fortuna implementations lies in the possibility of exposing associated *seed files* to attackers. The data in Fortuna seed files is used to feed entropy to Fortuna through *refresh* calls when an RNG is not immediately available—for example, immediately after a system reboot and before the system's RNGs have recorded any unpredictable events. However, if an identical seed file is used twice, Fortuna will produce the same bit sequence twice. Seed files should therefore be erased after use to ensure they aren't reused.

Finally, if two Fortuna instances are in the same state because they're sharing a seed file (meaning the same data in the entropy pools, including C and K), then the *next* operation will return the same bits in both instances.

Cryptographic vs. Noncryptographic PRNGs

There are cryptographic and noncryptographic PRNGs. Noncrypto PRNGs are designed to produce uniform distributions for applications such as scientific simulations or video games. However, you should never use noncrypto PRNGs in crypto applications, because they're insecure; they're concerned only with the quality of the bits' probability distribution and not with their predictability. Crypto PRNGs, on the other hand, are unpredictable because they're also concerned with the strength of the underlying *operations* used to deliver well-distributed bits.

Unfortunately, most PRNGs exposed by programming languages—such as libc's `rand` and `drand48`, PHP's `rand` and `mt_rand`, Python's `random` module, and Java's `java.util.Random` class—are noncryptographic. Defaulting to a noncrypto PRNG is a recipe for disaster because it often ends up being used in crypto applications, so be sure to use only crypto PRNGs when generating randomness related to cryptographic or security applications.

A Popular Noncrypto PRNG: Mersenne Twister

The *Mersenne Twister (MT)* algorithm is a noncryptographic PRNG used (at the time of this writing) in PHP, Python, R, Ruby, and many other systems. It's even been used (unfortunately) in blockchain wallet key generators. MT generates uniformly distributed random bits without statistical bias, but it's predictable: given a few bits produced by MT, one can guess which bits will follow.

Let's look under the hood to see what makes the Mersenne Twister insecure. The MT algorithm is much simpler than that of crypto PRNGs: its internal state is an array, S, consisting of 624 32-bit words. This array is initially set to $S_1, S_2, \ldots, S_{624}$ and evolves to $S_2, \ldots, S_{625}$, then $S_3, \ldots, S_{626}$, and so on, according to this equation:

$$S_{k+624} = S_{k+397} \oplus \mathbf{A}\left(\left(S_k \wedge \text{0x80000000}\right) \vee \left(S_{k+1} \wedge \text{0x7fffffff}\right)\right)$$

Here, $\oplus$ denotes the bitwise XOR (^ in the C programming language), $\wedge$ denotes the bitwise AND (& in C), $\vee$ denotes the bitwise OR (| in C), and $\mathbf{A}$ is a function that transforms some 32-bit word, x, to $(x \gg 1)$ if x's most significant bit is 0, or to $(x \gg 1) \oplus$ 0x9908b0df otherwise.

In this equation, bits of S interact with each other only through XORs. The operators $\wedge$ and $\vee$ never combine 2 bits of S together but instead combine bits of S with bits from the constants 0x80000000 and 0x7fffffff. This way, any bit from S_{625} can be expressed as an XOR of bits from S_{398}, S_1, and S_2, and any bit from any future state can be expressed as an XOR combination of bits from the initial state $S_1, \ldots, S_{624}$. (When you express, say, $S_{228+624} = S_{852}$ as a function of S_{625}, S_{228}, and S_{229}, you can in turn replace S_{625} by its expression in terms of S_{398}, S_1, and S_2.)

Because there are exactly $624 \times 32 = 19{,}968$ bits in the initial state (or 624 32-bit words), any output bit can be expressed as an equation with at most 19,969 terms (19,968 bits plus one constant bit). That's about 2.5KB of

data. The converse is also true: bits from the initial state can be expressed as an XOR of output bits.

Linearity Insecurity

We call an XOR combination of bits a *linear combination*. For example, if X, Y, and Z are bits, then the expression $X \oplus Y \oplus Z$ is a linear combination, whereas $(X \wedge Y) \oplus Z$ is not because there's an AND ($\wedge$). If you flip a bit of X in $X \oplus Y \oplus Z$, then the result changes as well, regardless of Y's and Z's values. In contrast, if you flip a bit of X in $(X \wedge Y) \oplus Z$, the result changes only if Y's bit at the same position is 1. The upshot is that linear combinations are predictable because you don't need to know the value of the bits in order to predict how a change in their value will affect the result.

For comparison, if the MT algorithm were cryptographically strong, its equations would be *nonlinear* and would involve not only single bits but also AND combinations (*products*) of bits, such as $S_1S_{15}S_{182}$ or $S_{17}S_{256}S_{257}S_{354}S_{498}S_{601}$. Although linear combinations of those bits include at most 624 variables, nonlinear combinations allow for up to 2^{624} variables. It would be impossible to solve, let alone write down, the whole of these equations. (Note that 2^{305}, a much smaller number, is the estimated information capacity of the observable universe.)

The key here is that linear transformations lead to short equations (comparable in size to the number of variables), which are easy to solve, whereas nonlinear transformations give rise to equations of exponential size, which are practically unsolvable. The game of cryptographers is thus to design PRNG algorithms that emulate such complex nonlinear transformations using only a small number of simple operations.

NOTE *Linearity is just one of many security criteria. Although necessary, nonlinearity alone does not make a PRNG cryptographically secure.*

The Uselessness of Statistical Tests

Statistical test suites like TestU01, Diehard, or the National Institute of Standards and Technology (NIST) test suite are one way to test the quality of pseudorandom bits. These tests take a sample of pseudorandom bits produced by a PRNG (say, 1MB's worth), compute some statistics on the distribution of certain patterns in the bits, and compare the results with the typical results obtained for a uniform distribution. For example, some tests count the number of 1 bits versus the number of 0 bits, or the distribution of 8-bit patterns. But statistical tests are largely irrelevant to cryptographic security, and it's possible to design a cryptographically weak PRNG that fools any statistical test.

When you run statistical tests on randomly generated data, you will usually see a bunch of statistical indicators as a result. These are typically *p*-values, a common statistical indicator. These results aren't always easy to interpret because they're rarely as simple as passed or failed. If your first results seem abnormal, don't worry: they may be the result of some

accidental deviation, or you may be testing too few samples. To ensure that the results you see are normal, compare them with those obtained for some reliable sample of identical size—for example, one generated with the OpenSSL toolkit using the following command:

```
$ openssl rand <number of bytes> -out <output file>
```

Real-World PRNGs

Let's turn our attention to implementing PRNGs in the real world. You'll find crypto PRNGs in the operating systems (OSs) of most platforms, from desktops and laptops to embedded systems such as routers and set-top boxes, as well as virtual machines, mobile phones, and so on. Most of these PRNGs are software based, but those that are pure hardware are used by applications running on the OS and sometimes by other PRNGs running on top of cryptographic libraries or applications.

Next, we'll look at the most widely deployed PRNGs: for Linux, Android, and many other Unix-based systems; in Windows; and in recent Intel microprocessors, whose PRNG is hardware based.

Random Bits in Linux

The device file */dev/urandom* is the userland interface to the crypto PRNG in operating systems based on the Linux kernel. You'll typically use it to generate reliable random bits. Because it's a device file, you request random bits from */dev/urandom* by reading it as a file. For example, the following command uses */dev/urandom* to write 10MB of random bits to a file:

```
$ dd if=/dev/urandom of=<output file> bs=1M count=10
```

The Wrong Way to Use /dev/urandom

You could write a naive and insecure C program like the one shown in Listing 2-1 to read random bits and hope for the best, but that would be a bad idea.

```
int random_bytes_insecure(void *buf, size_t len)
{
    int fd = open("/dev/urandom", O_RDONLY);
    read(fd, buf, len);
    close(fd);
    return 0;
}
```

Listing 2-1: An insecure use of /dev/urandom

This code is insecure; it doesn't even check the return values of `open()` and `read()`, which means your expected random buffer could end up filled with zeros or left unchanged.

A Safer Way to Use /dev/urandom

Listing 2-2, copied from the LibreSSL library, shows a safer way to use */dev/urandom*.

```
int random_bytes_safer(void *buf, size_t len)
{
    struct stat st;
    size_t i;
    int fd, cnt, flags;
    int save_errno = errno;

start:
    flags = O_RDONLY;
#ifdef O_NOFOLLOW
    flags |= O_NOFOLLOW;
#endif
#ifdef O_CLOEXEC
    flags |= O_CLOEXEC;
#endif
  ❶ fd = open("/dev/urandom", flags, 0);
    if (fd == -1) {
        if (errno == EINTR)
            goto start;
        goto nodevrandom;
    }
#ifndef O_CLOEXEC
    fcntl(fd, F_SETFD, fcntl(fd, F_GETFD) | FD_CLOEXEC);
#endif

    /* Lightly verify that the device node looks sane. */
    if (fstat(fd, &st) == -1 || !S_ISCHR(st.st_mode)) {
        close(fd);
        goto nodevrandom;
    }
    if (ioctl(fd, RNDGETENTCNT, &cnt) == -1) {
        close(fd);
        goto nodevrandom;
    }
    for (i = 0; i < len; ) {
        size_t wanted = len - i;
      ❷ ssize_t ret = read(fd, (char *)buf + i, wanted);

        if (ret == -1) {
            if (errno == EAGAIN || errno == EINTR)
                continue;
            close(fd);
            goto nodevrandom;
        }
        i += ret;
    }
    close(fd);
    if (gotdata(buf, len) == 0) {
        errno = save_errno;
        return 0; /* Satisfied */
    }
```

```
nodevrandom:
    errno = EIO;
    return -1;
}
```

Listing 2-2: A safe use of /dev/urandom

Unlike Listing 2-1, Listing 2-2 makes several sanity checks. Compare, for example, the calls to `open()` ❶ and to `read()` ❷ with those in Listing 2-1: the safer code checks the return values of those functions and upon failure closes the file descriptor and returns –1.

Differences Between /dev/urandom and /dev/random, Before 2022

The Linux PRNG, defined in *drivers/char/random.c* in the Linux kernel, underwent major changes in 2022 (since kernel version 5.17).

First, the general structure of the PRNG, which is similar in the old and new versions, is based on a collection of entropy from various sources (including system activity, such as keyboard, mouse, and disk accesses), as well as from an entropy pool that can be seen as a large array, which is filled by hashing data collected from the entropy sources. Next, a DRBG is responsible for producing the pseudorandom data streams returned when */dev/random* or */dev/urandom* is read or when the `getrandom()` system call is made.

Historically, prior to kernel version 5.17, the Linux PRNG behaved as follows: unlike */dev/urandom*, the */dev/random* interface was *blocking*; if the kernel estimated that the PRNG had an insufficient level of entropy, then */dev/random* would stop returning bytes ("block") when it was read, until a sufficient level of entropy was estimated by the kernel. This was not a good idea. For one thing, entropy estimators are notoriously unreliable and can be fooled by attackers (which is one reason why Fortuna ditched Yarrow's entropy estimation). Furthermore, */dev/random* ran out of estimated entropy pretty quickly, which could produce a denial-of-service condition, slowing applications that were forced to wait for more entropy. The upshot is that in practice, */dev/random* was no better than */dev/urandom* and created more problems than it solved.

Differences Between /dev/urandom and /dev/random, Since 2022

In versions of the Linux kernel from 2022 (5.17 onward), several improvements have been incorporated. First, the SHA-1 hash function has been replaced by BLAKE2 when creating the contents of the pool. The biggest change is the modification of the relative behavior of */dev/random* and */dev/urandom*; it has even been proposed to eliminate their differences altogether. At the time of writing, on most platforms both interfaces will detect if there isn't enough entropy, but */dev/urandom* will resume producing pseudorandom bits if the kernel fails to collect enough entropy, whereas */dev/random* will block.

In addition, the kernel's entropy estimation logic has been greatly improved: instead of considering that entropy decreases when PRNG bits are read (a cryptographic nonsense), the kernel just looks for the point when enough uncertainty (that is, entropy) has been collected—for example, at system startup.

You can read the entropy value of a Linux system in the */proc/sys/kernel/random/entropy_avail* file. In older versions of the kernel, this value was a maximum of 4,096 bits and decreased with the generation of PRNG bits. In the new kernels, the value is capped at 256 bits and therefore no longer decreases.

The CryptGenRandom() Function in Windows

In Windows, the legacy userland interface to the system's PRNG is the `CryptGenRandom()` function from the Cryptography application programming interface (API). Recent Windows versions replace the `CryptGenRandom()` function with the `BcryptGenRandom()` function in the Cryptography API: Next Generation (CNG). The Windows PRNG takes entropy from the kernel mode driver *cng.sys* (formerly *ksecdd.sys*), whose entropy collector is loosely based on Fortuna. As is usually the case in Windows, the process is complicated.

Listing 2-3 shows a typical C++ invocation of `CryptGenRandom()` with the required checks.

```
int random_bytes(unsigned char *out, size_t outlen)
{
    static HCRYPTPROV handle = 0; /* Only freed when the program ends */
    if(!handle) {
        if(!CryptAcquireContext(&handle, 0, 0, PROV_RSA_FULL,
                                CRYPT_VERIFYCONTEXT | CRYPT_SILENT)) {
            return -1;
        }
    }
    while(outlen > 0) {
        const DWORD len = outlen > 1048576UL ? 1048576UL : outlen;
        if(!CryptGenRandom(handle, len, out)) {
            return -2;
        }
        out    += len;
        outlen -= len;
    }
    return 0;
}
```

Listing 2-3: Using the Windows `CryptGenRandom()` PRNG interface

Prior to calling the actual PRNG, you need to declare a *cryptographic service provider* (`HCRYPTPROV`) and then acquire a *cryptographic context* with `CryptAcquireContext()`, which increases the likelihood that things will go wrong. For instance, the final version of the TrueCrypt encryption software was found to call `CryptAcquireContext()` in a way that could silently fail, leading to suboptimal randomness without notifying the user. Fortunately, the

newer and simpler `BCryptGenRandom()` interface for Windows doesn't require the code to explicitly open a handle (or at least makes it much easier to use without a handle).

A Hardware-Based PRNG: Intel Secure Key

We've discussed only software PRNGs so far, so let's take a look at a hardware one. The *Intel Digital Random Number Generator*, or *Intel Secure Key*, is a hardware PRNG introduced in 2012 in Intel's Ivy Bridge microarchitecture. It's based on NIST's SP 800-90 guidelines with the Advanced Encryption Standard (AES) in CTR_DRBG mode. Intel's PRNG is accessed through the `RDRAND` assembly instruction, which offers an interface independent of the operating system and is in principle faster than software PRNGs.

Whereas software PRNGs try to collect entropy from unpredictable sources, Intel Secure Key has a single entropy source that provides a serial stream of entropy data as zeros and ones. In hardware engineering terms, this entropy source is a dual differential jamb latch with feedback—essentially, a small hardware circuit that jumps between two states (0 or 1) depending on thermal noise fluctuations, at a frequency of 3 GHz. This is usually pretty reliable.

The `RDRAND` assembly instruction takes as an argument a register of 16, 32, or 64 bits and then writes a random value. When invoked, `RDRAND` sets the carry flag to 1 if the data set in the destination register is a valid random value, and to 0 otherwise; be sure to check the `CF` flag if you write assembly code directly. Note that the C intrinsics available in common compilers don't check the `CF` flag but do return its value.

NOTE *Intel's PRNG framework provides an assembly instruction other than `RDRAND`: the `RDSEED` assembly instruction returns random bits directly from the entropy source, after some conditioning or cryptographic processing. It's intended to be able to seed other PRNGs.*

Intel Secure Key is only partially documented, but it's built on known standards and has been audited by the well-regarded company Cryptography Research (see its report titled "Analysis of Intel's Ivy Bridge Digital Random Number Generator"). Nonetheless, there have been some concerns about its security, especially following Edward Snowden's revelations about cryptographic backdoors: PRNGs are indeed the perfect target for sabotage. If you're concerned but still want to use `RDRAND` or `RDSEED`, mix them with other entropy sources. Doing so will prevent effective exploitation of a hypothetical backdoor in Intel Secure Key's hardware or in the associated microcode in all but the most far-fetched scenarios.

How Things Can Go Wrong

To conclude, I'll present a few examples of randomness failures. There are countless examples to choose from, but I've chosen four that are simple enough to understand and illustrate different problems.

Poor Entropy Sources

In 1996, the SSL implementation of the Netscape browser was computing 128-bit PRNG seeds according to the pseudocode shown in Listing 2-4, copied from Goldberg and Wagner's page at *https://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html*.

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
  ❶ b = mklcpr(pid + seconds + (ppid << 12));
    seed = MD5(a, b); /* Derivation of a 128-bit value using the hash MD5 */

mklcpr(x) /* Not cryptographically significant; shown for completeness */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* A very good standard mixing function, source omitted */
```

Listing 2-4: Pseudocode of the Netscape browser's generation of 128-bit PRNG seeds

The problem here is that the PIDs and microseconds are guessable values. Assuming that you can guess the value of `seconds`, `microseconds` has only 10^6 possible values and thus an entropy of $\log(10^6)$, or about 20 bits. The process ID (PID) and parent process ID (PPID) are 15-bit values, so you'd expect 15 + 15 = 30 additional entropy bits. But looking at how `b` is computed ❶ shows that the overlap of 3 bits yields an entropy of about 15 + 12 = 27 bits, for a total entropy of only 47 bits, whereas a 128-bit seed should have 128 bits of entropy.

Insufficient Entropy at Boot Time

In 2012, researchers scanned the internet and harvested public keys from TLS certificates and SSH hosts. They found that a handful of systems had identical public keys, and in some cases very similar keys (namely, RSA keys with shared prime factors)—in short, two numbers, $n = pq$ and $n' = p'q'$, with $p = p'$, whereas normally all ps and qs should be different in distinct modulus values.

It turned out that many devices generated their public key early, at first boot, before having collected enough entropy, despite using an otherwise-decent PRNG (typically */dev/urandom*). PRNGs in different systems produced identical random bits due to having the same entropy source (for example, a hardcoded seed).

At a high level, the presence of identical keys is due to key-generation schemes like the following, in pseudocode:

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
n = p*q
```

If two systems run this code given an identical seed, they'll produce the same p, the same q, and therefore the same n.

The presence of shared primes in different keys is due to key-generation schemes where additional entropy is injected during the process, as demonstrated here:

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_entropy()
q = prng.generate_random_prime()
n = p*q
```

If two systems run this code with the same seed, they'll produce the same p, but the injection of entropy through `prng.add_entropy()` will ensure distinct qs.

The problem with shared prime factors is that given $n = pq$ and $n' = pq'$, it's trivial to recover the shared p by computing the greatest common divisor (GCD) of n and n'. For details, see the paper "Mining Your Ps and Qs" by Heninger, Durumeric, Wustrow, and Halderman, available at *https://factorable.net*.

Noncryptographic PRNG

Earlier we discussed the difference between crypto and noncrypto PRNGs and why the latter should never be used for crypto applications. Alas, many systems overlook that detail, so we'll look at one such example.

The popular MediaWiki application runs on Wikipedia and many other wikis. It uses randomness to generate things like security tokens and temporary passwords, which should be unpredictable. Unfortunately, a now obsolete version of MediaWiki used a noncrypto PRNG, the Mersenne Twister, to generate these tokens and passwords. Here's a snippet from the vulnerable MediaWiki source code; look for the function called to get a random bit, and read the comments:

```
        /**
         * Generate a hex-y looking random token for various uses.
         * Could be made more cryptographically sure if someone cares.
         * @return string
         */
function generateToken( $salt = '' ) {
    $token = dechex(mt_rand()).dechex(mt_rand());
    return md5( $token . $salt );
}
```

Did you notice `mt_rand()` in the preceding code? Here, `mt` stands for Mersenne Twister. In 2012, researchers showed how to exploit the predictability of Mersenne Twister to predict future tokens and temporary passwords, given a couple of security tokens. MediaWiki was patched to use a crypto PRNG.

Sampling Bug with Strong Randomness

The next bug shows how even a strong crypto PRNG with sufficient entropy can produce a biased distribution. The chat program Cryptocat was designed to offer secure communication. It used a function that attempted to create a uniformly distributed string of decimal digits—namely, numbers in the range 0 through 9. However, just taking random bytes modulo 10 doesn't yield a uniform distribution; when taking all numbers between 0 and 255 and reducing them modulo 10, you don't get an equal number of values in 0 to 9.

Cryptocat did the following to address that problem and obtain a uniform distribution:

```
Cryptocat.random = function() {
    var x, o = '';
    while (o.length < 16) {
        x = state.getBytes(1);
        if (x[0] <= 250) {
            o += x[0] % 10;
        }
    }
    return parseFloat('0.' + o)
}
```

And that was almost perfect. By taking only the numbers up to a multiple of 10 and discarding others, you'd expect a uniform distribution of the digits 0 through 9. Unfortunately, there was an off-by-one error in the `if` condition. I'll leave the details to you as an exercise. You should find that there is a small statistical bias in favor of the index 0 (hint: `<=` should have been `<`).

Further Reading

I've just scratched the surface of randomness in cryptography. There is much more to learn about the theory of randomness, including different entropy notions, randomness extractors, and even the power of randomization and derandomization in complexity theory. To learn more about PRNGs and their security, read the classic 1998 paper "Cryptanalytic Attacks on Pseudorandom Number Generators" by Kelsey, Schneier, Wagner, and Hall. Then look at the implementation of PRNGs in your favorite applications and try to find their weaknesses. (Search online for "random generator bug" to find plenty of examples.)

We're not done with randomness, though. We'll encounter it multiple times throughout this book, and you'll discover the many ways it helps to construct secure systems.

3

CRYPTOGRAPHIC SECURITY

Cryptographic definitions of security are not the same as those that apply to general computer security. The main difference between software security and cryptographic security is that we can *quantify* the latter. Unlike in the software world, where we usually say applications are either secure or insecure, in the cryptographic world it's often possible to calculate the amount of effort required to break a cryptographic algorithm. Also, whereas software security focuses on preventing attackers from abusing a program's code, the goal of cryptographic security is to make well-defined problems impossible to solve.

Cryptographic problems involve mathematical notions but not complex math—at least, not in this book. This chapter walks through some of these security notions and how you can apply them to solve real-world problems.

In the following sections, I discuss how to quantify crypto security in ways that are both theoretically sound and practically relevant. I discuss the notions of unconditional versus computational security, bit security versus full attack cost, provable versus heuristic security, and symmetric versus asymmetric key generation. I conclude the chapter with real-world examples of failures in seemingly strong cryptography.

Defining the Impossible

In Chapter 1, I described a cipher's security relative to an attacker's capabilities and goals and deemed a cipher secure if it's impossible to reach these goals given an attacker's known capabilities. But what does *impossible* mean in this context?

Two notions define the concept of impossible in cryptography: unconditional security and computational security. Roughly speaking, *unconditional security* is about theoretical impossibility, whereas *computational security* is about practical impossibility. Unconditional security doesn't quantify security because it views a cipher as either secure or insecure, with no middle ground; it's therefore useless in practice, although it plays an important role in theoretical cryptography. Computational security is the more relevant and practical measure of the strength of a cipher.

Security in Theory: Unconditional Security

Unconditional security is based not on how hard it is to break a cipher but on whether it's conceivable to break it at all. A cipher is unconditionally secure only if, given unlimited computation time and memory, it cannot be broken. Even if a successful attack on a cipher would take trillions of years, such a cipher is unconditionally *in*secure.

For example, the one-time pad in Chapter 1 is unconditionally secure. Recall that the one-time pad encrypts a plaintext, P, to a ciphertext, $C = P \oplus K$, where K is a random bit string that is unique to each plaintext. The cipher is unconditionally secure because, given a ciphertext and unlimited time to try all possible keys, K, and compute the corresponding plaintext, P, you'd still be unable to identify the right K because there are as many possible Ps as there are Ks.

Security in Practice: Computational Security

Unlike unconditional security, computational security views a cipher as secure if it cannot be broken within a *reasonable* amount of time and with reasonable resources such as memory, hardware, budget, and energy. Computational security is a way to quantify the security of a cipher or any crypto algorithm.

For example, consider a cipher, $\mathbf{E}$, for which you know a plaintext–ciphertext pair (P, C) but not the 128-bit key, K, that computes $C = \mathbf{E}(K, P)$. This cipher is not unconditionally secure because you could break it after trying the 2^{128} possible 128-bit Ks until you find the one that satisfies

$\mathbf{E}(K, P) = C$. But in practice, even with testing 100 billion keys per second, it would take more than 100,000,000,000,000,000,000 years. In other words, reasonably speaking, this cipher is computationally secure because it's practically impossible to break.

We can express computational security in terms of two values:

- t, which is a limit on the number of operations that an attacker will carry out
- ε (*epsilon*), which is a limit on the probability of success of an attack

We then say that a cryptographic scheme is (t, ε)-*secure* if an attacker performing at most t operations—whatever those operations are—has a probability of success that is no higher than ε, where ε is at least 0 and at most 1. Computational security gives a limit on how hard it is to break a cryptographic algorithm.

Recognize that t and ε are just limits: if a cipher is (t, ε)-secure, then no attacker performing fewer than t operations will succeed (with probability ε). However, this doesn't imply that an attacker doing exactly t operations will succeed, and it doesn't provide the necessary number of operations, which may be much larger than t. We say that t is a *lower bound* on the necessary computation effort because you'd need at least t operations to compromise security.

If we do know precisely how much effort it takes to break a cipher, we say that (t, ε)-security gives us a *tight bound* when an attack exists that breaks the cipher with probability ε and exactly t operations.

For example, consider a symmetric cipher with a 128-bit key. Ideally, this cipher should be $(t, t/2^{128})$-secure for any value of t between 1 and 2^{128}. The best attack should be *brute force* (trying all keys until you find the correct one). Any better attack would have to exploit some imperfection in the cipher, so we strive to create ciphers where brute force is the best possible attack.

Given the statement $(t, t/2^{128})$-secure, let's examine the probability of success of three possible attacks:

- In the first case, $t = 1$, an attacker tries one key and succeeds with a probability of $\varepsilon = 1/2^{128}$.
- In the second case, $t = 2^{128}$, an attacker tries all 2^{128} keys, and one succeeds. Thus, the probability $\varepsilon = 1$. (If the attacker tries all keys, the right one must be among them.)
- In the third case, an attacker tries only $t = 2^{64}$ keys and succeeds with a probability of $\varepsilon = 2^{64}/2^{128} = 2^{-64}$. When an attacker tries only a fraction of all keys, the success probability is proportional to the number of keys tried.

We can conclude that a cipher with a key of n bits is at best $(t, t/2^n)$-secure, for any t between 1 and 2^n, because no matter how strong the cipher, a brute-force attack against it will always succeed. The key thus needs to be long enough to blunt brute-force attacks in practice.

NOTE *In this example, we're counting the number of evaluations of the cipher, not the absolute time or number of processor clock cycles. Computational security is technology agnostic, which means a cipher that's* (t, ε)*-secure today will be* (t, ε)*-secure tomorrow—but what's considered secure in practice today might not be considered secure tomorrow.*

Quantifying Security

When you've found an attack, you should first figure out how efficient it is in theory and how practical it is, if at all. Likewise, given a cipher that's allegedly secure, you'll want to know what amount of work it can withstand. To address those questions, I'll explain how we measure cryptographic security in bits (the theoretical view) and what factors affect the actual cost of an attack.

Measuring Security in Bits

When speaking of computational security, a cipher is *t*-secure when a successful attack needs at least *t* operations. We thus avoid the nonintuitive (t, ε) notation by assuming a success probability of ε close to 1, or whatever probability we care about in practice. We then express security in bits, where "*n*-bit security" means that we need about 2^n operations to compromise some particular security notion.

If you know approximately how many operations it takes to break a cipher, you can determine its security level in bits by taking the binary logarithm of the number of operations: if it takes 1,000,000 operations, the security level is $\log_2(1{,}000{,}000)$, or about 20 bits (as 1,000,000 is approximately equal to 2^{20}). Recall that an *n*-bit key will give at most *n*-bit security because a brute-force attack with all 2^n possible keys will always succeed. But the key size doesn't always match the security level—it just gives an *upper bound*, or the highest possible security level.

A security level may be smaller than the key size for one of two reasons:

- An attack broke the cipher in fewer operations than expected—for example, using a method that recovers the key by trying only a subset of the 2^n keys.
- The cipher's security level intentionally differs from its key size, as with most public-key algorithms. For example, the RSA algorithm with 1,024-bit private-key elements (thus with a 2,048-bit modulus) provides less than 128-bit security.

Bit security proves useful when comparing the security level of ciphers, but it doesn't provide enough information on the actual cost of an attack. It's sometimes too simple an abstraction because it assumes that an *n*-bit-secure cipher takes 2^n operations to break, whatever these operations are. Two ciphers with the same bit security level can therefore have vastly different real-world security levels when you factor in the actual cost of an attack to a real attacker.

Say we have two ciphers, each with a 128-bit key and 128-bit security. We must evaluate each cipher 2^{128} times to break it, but the second cipher is 100 times slower than the first. Evaluating the second cipher 2^{128} times thus takes the same time as $100 \times 2^{128} \approx 2^{134.64}$ evaluations of the first. If we count in terms of the first, fast cipher, then breaking the slower one takes $2^{134.64}$ operations. If we count in terms of the second, slow cipher, it takes only 2^{128} operations. Should we then say that the second cipher is stronger than the first? In principle, yes, but we rarely see such a hundred-fold performance difference between common ciphers.

The inconsistent definition of an operation raises more difficulties when comparing the efficiency of attacks. Some attacks claim to reduce a cipher's security because they perform 2^{120} evaluations of some operation rather than 2^{128} evaluations of the cipher, but the speed of each type of attack is left out of the analysis. The 2^{120}-operation attack won't always be faster than a 2^{128} brute-force attack.

Nevertheless, bit security remains a useful notion as long as the operation is reasonably defined—meaning about as fast as an evaluation of the cipher. After all, in real life, all it takes to determine whether a security level is sufficient is an order of magnitude.

Calculating the Full Attack Cost

Bit security expresses the cost of the fastest attack against a cipher by estimating the order of magnitude of the number of operations it needs to succeed. But other factors affect the cost of an attack, and we must take these into account when estimating the actual security level. I'll explain the four main ones: parallelism, memory, precomputation, and the number of targets.

Parallelism

The first factor to consider is computational parallelism—that is, the ability of the attack's implementation to take advantage of parallel computing, such as multicore systems.

For example, consider these two attacks of 2^{56} operations each:

- The first attack performs 2^{56} *sequentially dependent* operations, computing $x_{i+1} = f_i(x_i)$ for some fixed x_0 and distinct functions f_i (with i from 1 to 2^{56}).
- The second attack performs 2^{56} *independent* operations, computing $x_i = f_i(x)$ for some fixed x and distinct functions f_i (with i from 1 to 2^{56}). It can execute the $f_i(x)$ computations in parallel because each $f_i(x)$ is independent of the others.

The difference between the two attacks is that one can parallelize the second attack but not the first. Parallel processing can be orders of magnitude faster than sequential processing. For example, if you had $2^{16} = 65{,}536$ processors available, you could divide the workload of the parallel attacks into 2^{16} independent tasks, each performing $2^{56} / 2^{16} = 2^{40}$ operations. The first attack, however, cannot benefit from having multiple cores available

because each operation relies on the previous operation's result. Therefore, the parallel attack will complete 65,536 times faster than the sequential one, even though they perform the same number of operations.

NOTE *Algorithms that become* N *times faster to attack when* N *cores are available are* embarrassingly parallel*; their execution times scale linearly with respect to the number of computing cores.*

Memory

The second factor when determining the cost of an attack is memory. We evaluate cryptanalytic attacks with respect to their use of time and space: How many operations do they perform over time, how much memory or space do they consume, how do they use the space they consume, and what's the speed of the available memory? Unfortunately, bit security focuses only on the time it takes to perform an attack.

Concerning the way an attack uses space, it's important to consider how many memory lookups the attack requires, the speed of memory accesses (which may differ between reads and writes), the size of the accessed data, the access pattern (contiguous or random memory addresses), and how it structures data in memory. For example, on a 2021 Intel Xeon 8380 Ice Lake processor, accessing a register takes 1 clock cycle, accessing the L1 cache (48kB) takes 5 cycles, accessing the L2 cache (1.25MB) takes 14 cycles, accessing the L3 cache (60MB) takes 63.5 cycles, and accessing DRAM is at best as fast but often much slower than accessing the L3 cache (the exact delay depends on several factors).

Precomputation

Precomputation operations need to be performed only once and can be reused over subsequent executions of the attack. We sometimes call precomputation the *offline stage* of an attack.

Consider the time-memory trade-off attack, in which the attacker performs one huge computation that produces large lookup tables that we then store and reuse to perform the actual attack. For example, one attack on 2G mobile encryption took two months to build 2TB's worth of tables, which attackers then used to break the encryption in 2G and recover a secret session key in only a few seconds.

Number of Targets

Finally, we come to the number of targets of the attack. The greater the number of targets, the greater the attack surface, and the more attackers can learn about the keys they're after.

For example, consider a brute-force key search: if you target a single n-bit key, it will take 2^n attempts to find the correct key with certainty. If you target multiple n-bit keys—say, a number M—and if for a single P you have M distinct ciphertexts, where $C = \mathbf{E}(K, P)$ for each of the M keys (K) that you're after, it will again take 2^n attempts to find each key. But if you're

interested only in *at least one* of the M keys and not in every one, it would take on average $2^n/M$ attempts to succeed. For example, to break one 128-bit key of $2^{16} = 65{,}536$ target keys, it will take on average $2^{128-16} = 2^{112}$ evaluations of the cipher. That is, the cost (and speed) of the attack decreases as the number of targets increases.

Choosing and Evaluating Security Levels

Choosing a security level often involves selecting between the 128-bit and 256-bit security levels available in most standard crypto algorithms and implementations. You'll find schemes with 64- or 80-bit security, but these are generally not secure enough for real-world use.

At a high level, 128-bit security means you'd need to carry out approximately 2^{128} operations to break that cryptosystem. To give you a sense of what this number means, consider the fact that the universe is approximately 2^{88} nanoseconds old (there's a billion nanoseconds in a second). Since testing a key with today's technology takes no less than a nanosecond, you'd need several times the age of the universe for an attack to succeed (2^{40} times, to be precise).

But can't parallelism and multiple targets dramatically reduce the time it takes to complete a successful attack? Not exactly. Say you're interested in breaking any one of a million targets and that you have a million parallel cores available. That brings the search time down from 2^{128} to $(2^{128} / 2^{20}) / 2^{20} = 2^{88}$, which is equivalent to "only" one universe lifetime.

Another thing to consider when evaluating security levels is the evolution of technology. Moore's law posits that computing efficiency doubles roughly every two years. We can think of this as a loss of 1 bit of security every two years: if today a $1,000 budget allows you to break, say, a 40-bit key in one hour, then Moore's law says that two years later, you could break a 41-bit key in one hour for the same $1,000 budget (I'm simplifying). We can extrapolate from this to say that, according to Moore's law, we'll have 40 fewer bits of security in 80 years compared to today. In other words, in 80 years doing 2^{128} operations may cost as much as doing 2^{88} operations today. Accounting for parallelism and multiple targets, we're down to 2^{48} nanoseconds of computation, or about three days. But this extrapolation is highly inaccurate because Moore's law won't and can't scale that much. Still, you get the idea: what looks infeasible today may be realistic in a century.

There will be times when a security level lower than 128 bits is justified such as when you need security for a short time period and when the costs of implementing a higher security level will negatively impact the cost or usability of a system. An example is pay-TV systems, wherein encryption keys are either 48 or 64 bits. This sounds ridiculously low but is sufficient because the key refreshes every 5 or 10 seconds.

Nevertheless, to ensure long-term security, you should choose 256-bit security or a bit less. Even in a worst-case scenario—the existence of quantum computers (see Chapter 14)—we're unlikely to break a 256-bit secure scheme in the foreseeable future. More than 256 bits of security is practically unnecessary, except as a marketing device.

As cryptographer John Kelsey once put it: "The difference between 80 bits and 128 bits of key search is like the difference between a mission to Mars and a mission to Alpha Centauri. As far as I can see, there is no meaningful difference between 192-bit and 256-bit keys in terms of practical brute-force attacks; impossible is impossible."

Achieving Security

Once you've chosen a security level, it's important to guarantee that your cryptographic schemes will stick to it. In other words, you want *confidence*, not just hope and uncertainty, that things will work as planned, all the time.

When building confidence in the security of a crypto algorithm, you can rely on mathematical proofs, an approach we call *provable security*, or on evidence of failed attempts to break the algorithm, which I'll call *heuristic security* (though it's sometimes called *probable* security). These two approaches are complementary, and neither is better than the other, as you'll see.

Provable Security

Provable security is about proving that breaking your crypto scheme is at least as hard as solving another problem known to be hard. Such a *security proof* guarantees that the crypto remains safe as long as the hard problem remains hard. This type of proof is called a *reduction*, and it comes from the field of complexity theory. We say problem X is reducible to breaking some cipher if any method to break the cipher also yields a method to solve problem X. Such a reduction guarantees that as long as problem X is hard, the cipher is secure.

Security proofs come in two flavors, depending on the type of presumably hard problem you use: proofs relative to a mathematical problem and proofs relative to a cryptographic problem.

Proofs Relative to a Mathematical Problem

Many security proofs (such as those for public-key crypto) show that breaking a crypto scheme is at least as hard as solving some hard mathematical problem. We're talking of problems for which we know a solution exists and is easy to verify once we know it but is computationally hard to find.

NOTE *There's no real proof that seemingly hard math problems are actually hard. In fact, proving this for a specific class of problems is one of the greatest challenges in the field of complexity theory. As I write this, there is a $1,000,000 bounty for anyone who can prove this, awarded by the Clay Mathematics Institute. I discuss this in more detail in Chapter 9.*

For example, consider the challenge of solving the *factoring problem*, which is the best-known math problem in crypto: given a number that you know is the product of two prime numbers ($n = pq$), find the said primes. For example, if $n = 15$, the answer is 3 and 5. That's easy for a small number,

but it becomes exponentially harder as the size of the number grows. For example, if a number, n, is 3,000 bits long (about 900 decimal digits) or more, factoring is believed to be practically infeasible.

Rivest–Shamir–Adleman (RSA) is the most famous crypto scheme to rely on the factoring problem: RSA encrypts a plaintext, P, seen as a large number, by computing $C = P^e \bmod n$, where the number e and $n = pq$ are the public key. Decryption recovers a plaintext from a ciphertext by computing $P = C^d \bmod n$, where d is the private key associated with e and n. If we can factor n, then we can break RSA (by recovering the private key from the public key), and if we can obtain the private key, then we can factor n (for example, see the article at *https://eprint.iacr.org/2004/208*). In other words, recovering an RSA private key and factoring n are equivalently hard problems. That's the kind of reduction we're looking for in provable security. However, there is no guarantee that recovering an RSA plaintext is as hard as factoring n, since the knowledge of a plaintext doesn't reveal the private key.

Proofs Relative to Another Crypto Problem

Instead of comparing a crypto scheme to a math problem, you can compare it to another crypto scheme and prove that you can break the second scheme only if you can break the first. Security proofs for symmetric ciphers usually follow this approach.

For example, if all you have is a single permutation algorithm, then you can build symmetric ciphers, random bit generators, and other crypto objects such as hash functions by combining calls to the permutations with various types of inputs (as you'll see in Chapter 6). Proofs then show that the newly created schemes are secure if the permutation is secure. In other words, we know that the newly created algorithm is *not weaker* than the original one. Such proofs usually work by crafting an attack on the smaller component, given an attack on the larger one—that is, by showing a reduction.

When proving that a crypto algorithm is no weaker than another, the main benefit is that of a reduced attack surface: instead of analyzing both the core algorithm and the combination, you can simply look at the new cipher's core algorithm. Specifically, if you write a cipher that uses a newly developed permutation and a new combination, you may prove that the combination doesn't weaken security compared to the core algorithm. Therefore, to break the combination, you need to break the new permutation.

Caveats

Cryptography researchers rely heavily on security proofs, whether with respect to math problem schemes or to other crypto schemes. But the existence of a security proof doesn't guarantee that a cryptographic scheme is perfect, nor is it an excuse for neglecting the more practical aspects of implementation. After all, as cryptographer Lars Knudsen once said, "If it's provably secure, it's probably not," meaning that a security proof shouldn't be taken as an absolute guarantee of security. Worse, there are multiple reasons why a "provably secure" scheme may lead to a security failure.

One issue is with the phrase "proof of security" itself. In mathematics, a proof is the demonstration of an *absolute truth*, but in crypto, a proof is only the demonstration of a *relative truth*. For example, a proof that your cipher is as hard to break as it is to compute discrete logarithms—finding the number x given g and $g^x \bmod n$—guarantees that if your cipher fails, a whole lot of other ciphers will fail as well, and nobody will blame you if the worst happens.

Another caveat is that one usually proves security with respect to a single notion of security. For example, you might prove that recovering the private key of a cipher is as hard as the factoring problem. But if you can recover plaintexts from ciphertext without the key, you'll bypass the proof, and recovering the key hardly matters.

Then again, proofs are not always correct, and it may be easier to break an algorithm than originally thought.

NOTE *Unfortunately, few researchers carefully check security proofs, which commonly span dozens of pages, thus complicating quality control. That said, demonstrating that a proof is incorrect doesn't necessarily imply that the proof's goal is completely wrong; if the result is correct, one may salvage the proof by correcting its errors.*

Another important consideration is that hard math problems sometimes turn out to be easier to solve than expected. For example, certain weak parameters make it easy to break the RSA cryptosystem. Or the math problem may be hard in certain cases but not on average, as often happens when the reference problem is new and not well understood. That's what happened when the 1978 knapsack encryption scheme by Merkle and Hellman was later broken using lattice reduction techniques.

Finally, although the proof of an algorithm's security may be fine, the implementation of the algorithm can be weak. For example, attackers may exploit side-channel information such as power consumption or execution time to learn about an algorithm's internal operations in order to break it, thus bypassing the proof. Or implementers may misuse the crypto scheme: if the algorithm is too complicated with too many knobs to configure, chances are higher that the user or developer will get a configuration wrong, which may render the algorithm completely insecure.

Heuristic Security

Provable security is a great tool to gain confidence in a crypto scheme, but it doesn't apply to all kinds of algorithms. In fact, most symmetric ciphers don't have a security proof. For example, every day we rely on AES to securely communicate using our mobile phones, laptops, and desktop computers, but AES is not provably secure; there's no proof that it's as hard to break as some well-known problem. AES can't be related to a math problem or to another algorithm because it's the hard problem itself.

In cases where provable security doesn't apply, the only reason to trust a cipher is because many skilled people tried to break it and failed. We call this *heuristic security*.

When can we be sure that a cipher is secure? We can never be sure, but we can be pretty confident that an algorithm won't be broken when hundreds of experienced cryptanalysts have each spent hundreds of hours trying to break it and published their findings—usually by attempting attacks on *simplified versions* of a cipher (often versions with fewer operations, or fewer *rounds*, which are short series of operations that ciphers iterate in order to mix bits together).

When analyzing a new cipher, cryptanalysts first try to break one round, then two, three, or as many as they can. The *security margin* is then the difference between the total number of rounds and the number of successfully attacked rounds. After years of study, if a cipher's security margin is still high, we become confident that it's (probably) secure.

Generating Keys

If you plan to encrypt something, you'll have to generate keys, whether they're temporary "session keys" (like the ones generated when browsing an HTTPS site) or long-term public keys. Recall from Chapter 2 that secret keys are the crux of cryptographic security and should be randomly generated so that they are unpredictable and secret.

For example, when you browse an HTTPS website, your browser receives the site's public key and uses it to establish a symmetric key that's valid only for the current session, and that site's public key and its associated private key may be valid for years. Therefore, it'd better be hard for an attacker to find. But generating a secret key isn't always as simple as dumping enough pseudorandom bits. We can generate cryptographic keys in one of three ways:

- *Randomly*, using a PRNG feeding a key-generation algorithm
- From a *password*, using a password-based key derivation function (PBKDF), which transforms the user-supplied password into a key
- Through a *key agreement protocol*, which is a series of message exchanges between two or more parties that ends with the establishment of a shared key

For now, I'll explain the simplest method: randomized generation.

Symmetric Keys

Symmetric keys are secret keys shared by two parties, and they are the simplest to generate. They are usually the same length as the security level they provide: a 128-bit key provides 128-bit security, and any of the 2^{128} possible keys is valid.

To generate a symmetric key of n bits using a cryptographic PRNG, you simply ask it for n pseudorandom bits and use those bits as the key.

That's it. You can, for example, use the OpenSSL toolkit to generate a random symmetric key by dumping pseudorandom bytes, as in the following command:

```
$ openssl rand -hex 16
65a4400ea649d282b855bd2e246812c6
```

Your result will, of course, differ from mine.

Asymmetric Keys

Unlike symmetric keys, asymmetric keys are usually longer than the security level they provide. But the main problem stems from asymmetric keys being trickier to generate than symmetric ones because you can't dump *n* bits from your PRNG and get away with the result. Asymmetric keys aren't just raw bit sequences. Instead, they represent a specific type of object, such as a large number with specific properties (in RSA, a product of two primes). A random bit string value (and thus a random number) is unlikely to have the necessary properties and therefore won't be a valid key.

To generate an asymmetric key, you send pseudorandom bits as a seed to a *key-generation algorithm*. This algorithm takes as input a seed value that's at least as long as the intended security level and constructs from it a private key and its respective public key, ensuring that both satisfy the necessary criteria. For example, a naive key-generation algorithm for RSA would generate a number, $n = pq$, by using an algorithm to generate two random primes of about the same length. That algorithm would pick random numbers until one happens to be prime, so you'd also need an algorithm to test whether a number is prime.

To save yourself the burden of manually implementing the key-generation algorithm, you can use OpenSSL to generate a 4,096-bit RSA private key, like this:

```
$ openssl genrsa 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....................................................................
.............................++
.........................................++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MIIJKQIBAAKCAgEA3Qgm6OjMy61YVstaGawk22A9LyMXhiQUU4N8F5QZXEef2Pjq
vTtAIA1hzpK2AJsv16INpNkYcTjNmechAJOxHraftO6cp2pZFP85dvknsMfUoe8u
btKXZiYvJwpSOfQQ4tzlDtH45Gj8sMHcwFxTO3HSIxOXVOowfJTLMzZbSE3TDlN+
JdW8d9Xd5UVB+o9gUCI8tSfnOjF2dHlLNiOhlfT4wORf+G35USIyUJZtOQODh8M+
--snip--
zO/dbYtqRkMT8Ubb/OQ1IWOq8eOWnFetzkwPzAIjwZGXTOkWJu3RYj1OXbTYDr2c
xBRVC/ujoDL6O3NaqPxkWY5HJVmkyKIE5pCO4RFNyaQ8+o4APyobabPMylQq5Vo5
N5L2c4mhy1/OH8fvKBRDuvCk2oZinjdoKUo8ZA5DOa4pdvIQfR+b4/4Jjsx4
-----END RSA PRIVATE KEY-----
```

Notice that the key comes in a specific format—namely, base64-encoded data between the `BEGIN RSA PRIVATE KEY` and `END RSA PRIVATE KEY` markers. That's a standard encoding format that most systems support and can convert to raw bytes of data. The dot sequences at the beginning are a kind of progress bar, and `e is 65537 (0x10001)` indicates the parameter to use when encrypting (remember that RSA encrypts by computing $C = P^e \mod n$).

Protecting Keys

Once you have a secret key, you need to keep it secret yet available when you need it. There are three ways to address this problem:

Key wrapping (encrypting the key using a second key)

The problem with this approach is that the second key must be available when you need to decrypt the protected key. In practice, this second key is often generated from a password the user supplies when they need to use the protected key. That's how private keys for the Secure Shell (SSH) protocol are usually protected.

On-the-fly generation from a password

This doesn't require storing an encrypted file because the key comes straight out from the password. Systems like cryptocurrency wallets and password managers often use this method.

Although this method is more direct than key wrapping, it's less widespread, in part because it's more vulnerable to weak passwords. Say, for example, that an attacker captured some encrypted message: if we used key wrapping, the attacker first needs to get the protected key file, which may be stored locally on the user's file system or in a key management system (KMS), and therefore is not easy to access. But if we used on-the-fly generation, the attacker can directly search for the correct password by attempting to decrypt the encrypted message with candidate passwords. And if the password is weak, the key is compromised.

Storing the key on a hardware token (smart card or USB dongle)

In this approach, we store the key in secure memory, and it remains safe even if the computer is compromised. This is the safest approach to key storage but also the costliest and least convenient because it requires you to carry the hardware token with you and run the risk of losing it. Smart cards and USB dongles usually require you to enter a password to unlock the key from the secure memory.

NOTE *Whatever method you use, make sure not to mistake the private key for the public one when exchanging keys, and don't accidentally publish the private key through email or source code. (I've actually found private keys on GitHub.)*

To test key wrapping, run the following OpenSSL command with the argument `-aes128` to tell OpenSSL to encrypt the key with the cipher AES-128 (AES with a 128-bit key):

```
$ openssl genrsa -aes128 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
..........++
.....................................................................
.....................................................................
.....................................................++
e is 65537 (0x10001)
Enter pass phrase:
```

OpenSSL will use the requested passphrase to encrypt the newly created key.

How Things Can Go Wrong

Cryptographic security can go wrong in many ways. The biggest risk is when you have a false sense of security due to security proofs or well-studied protocols, as the following examples illustrate.

Incorrect Security Proof

Even proofs of security by renowned researchers may be wrong. One of the most striking examples of a proof gone terribly wrong is that of *Optimal Asymmetric Encryption Padding (OAEP)*, a method of secure encryption that used RSA and was implemented in many applications. An incorrect proof of OAEP's security against chosen-ciphertext attackers was accepted as valid for seven years, until a researcher found a flaw in 2001. Not only was the proof wrong, but the result was wrong as well. A new proof later showed that OAEP is only almost secure against chosen-ciphertext attackers. We now have to trust the new proof and hope that it's flawless. (For further details, see the 2001 paper "OAEP Reconsidered" by Victor Shoup.)

Short Keys for Legacy Support

In 2015, researchers found that some HTTPS sites and SSH servers supported public-key cryptography with shorter keys than expected—namely, 512 bits instead of at least 2,048 bits. Remember, with public-key schemes, the security level isn't equal to the key size, and in the case of HTTPS, keys of 512 bits offer a security level of approximately 60 bits. These keys could be broken after only about two weeks of computation using a cluster of 72 processors. This affected many websites, including the website of the US Federal Bureau of Investigation (FBI). Although the software was ultimately fixed (thanks to patches for OpenSSL and for other software), the problem was quite an unpleasant surprise.

Further Reading

To learn more about provable security for symmetric ciphers, read the sponge functions documentation (*https://keccak.team/sponge_duplex.html*). Sponge functions introduced the permutation-based approach in symmetric crypto, which describes how to construct a bunch of different cryptographic functions using only one permutation.

Some must-reads on the real cost of attacks include Daniel J. Bernstein's 2005 paper "Understanding Brute Force" and Michael Wiener's 2004 paper "The Full Cost of Cryptanalytic Attacks," both available online for free.

To determine the security level for a given key size, visit *https://www.keylength.com*. This site presents the recommendations of several government agencies concerning key sizes, as well as the order of magnitude of security guaranteed according to the size of public keys.

Finally, as an exercise, pick an application (such as a secure messaging application) and identify its crypto schemes, key length, and respective security levels. You'll often find surprising inconsistencies, such as a first scheme providing a 256-bit security level but a second scheme providing only 100-bit security. The security of the whole system is often only as strong as that of its weakest component.

PART II

SYMMETRIC CRYPTO

4

BLOCK CIPHERS

During the Cold War, the United States and the Soviets developed their own ciphers. The US government created the Data Encryption Standard (DES), which was adopted as a federal standard from 1979 to 2005, while the KGB developed GOST 28147-89, an algorithm kept secret until 1990 and still used today. In 2000, the US-based National Institute of Standards and Technology (NIST) selected the successor to DES, the Advanced Encryption Standard (AES), an algorithm developed in Belgium and now found in most electronic devices. AES, DES, and GOST 28147-89 are all *block ciphers*, a type of cipher that combines a core algorithm working on blocks of data with a *mode of operation*, or a technique to process sequences of data blocks.

This chapter reviews the core algorithms that underlie block ciphers, discusses their modes of operation, and explains how they all work together. It also discusses how AES works and concludes with coverage of a classic attack tool from the 1970s, the meet-in-the-middle attack, and a favorite attack technique of the 2000s—padding oracles.

What Is a Block Cipher?

A block cipher consists of an encryption algorithm and a decryption algorithm:

- The *encryption algorithm* (**E**) takes a key, K, and a plaintext block, P, and produces a ciphertext block, C. We write an encryption operation as $C = \mathbf{E}(K, P)$.
- The *decryption algorithm* (**D**) is the inverse of the encryption algorithm and decrypts a message to the original plaintext, P. We write this operation as $P = \mathbf{D}(K, C)$.

Since they're the inverse of each other, the encryption and decryption algorithms usually involve similar operations.

Security Goals

If you've followed earlier discussions about encryption, randomness, and indistinguishability, the definition of a secure block cipher will come as no surprise. We'll continue to define security as random-looking-ness, so to speak.

For a block cipher to be secure, it should be a *pseudorandom permutation (PRP)*, meaning that as long as the key is secret, an attacker shouldn't be able to compute an output of the block cipher from any input. That is, as long as K is secret and random from an attacker's perspective, they should have no clue what $\mathbf{E}(K, P)$ looks like, for any given P.

More generally, attackers should be unable to discover any *pattern* in the input/output values of a block cipher. In other words, it should be impossible to tell a block cipher from a truly random permutation, given black-box access to the encryption and decryption functions for some fixed and unknown key. By the same token, attackers should be unable to recover a secure block cipher's secret key; otherwise, they could use that key to tell the block cipher from a random permutation. This implies that attackers can't predict the plaintext that corresponds to a given ciphertext the block cipher produces.

Block Size

Two values characterize a block cipher: the block size and the key size. Security depends on both values. Most block ciphers have either 64-bit or 128-bit blocks—DES's blocks have 64 (2^6) bits, and AES's blocks have 128 (2^7) bits. In computing, lengths that are generally measured as powers

of two simplify data processing, storage, and addressing. But why 2^6 and 2^7 and not 2^4 or 2^{16} bits?

It's important that a block cipher's blocks aren't too large in order to minimize both the length of ciphertext and the memory footprint. Block ciphers first transform their input data into a sequence of blocks, which means that to encrypt a 16-bit message when blocks are 128 bits, you need to convert the message into a 128-bit block for the block cipher to process it and return a 128-bit ciphertext. The wider the blocks, the longer this overhead. To process a 128-bit block, you need at least 128 bits of memory. Blocks of 64, 128, or even 512 bits are short enough to fit in the registers of most CPUs or to implement using dedicated hardware circuits, allowing for efficient implementations in most cases. But larger blocks (for example, several kilobytes long) can have a noticeable impact on the cost and performance of implementations.

When ciphertexts' length or memory footprint is critical, you may have to use 64-bit blocks because these produce shorter ciphertexts and consume less memory. Otherwise, 128-bit or larger blocks are better, mainly because modern CPUs can often process 128-bit blocks more efficiently than 64-bit ones and they're more secure (see the "Sweet32" attack, at *https://sweet32.info*). In particular, CPUs can leverage instructions to efficiently process one or more 128-bit blocks in parallel—for example, the Advanced Vector Extensions (AVX) family of instructions in Intel CPUs.

The Codebook Attack

While blocks shouldn't be too large, they also shouldn't be too small; otherwise, they may be susceptible to *codebook attacks*, which are attacks against block ciphers that are efficient only when using smaller blocks. With 16-bit blocks, the codebook attack works like this:

1. Get the 65,536 (2^{16}) ciphertexts corresponding to each 16-bit plaintext block.
2. Build a lookup table—the *codebook*—mapping each ciphertext block to its corresponding plaintext block.
3. To decrypt an unknown ciphertext block, look up its corresponding plaintext block in the table.

When using 16-bit blocks, the lookup table needs only $2^{16} \times 16 = 2^{20}$ bits of memory, or 128 kilobytes. With 32-bit blocks, memory needs to grow to 16 gigabytes, which is still manageable. But with 64-bit blocks, you'd have to store 2^{70} bits (a zettabit, or 128 exabytes), so forget about it. Codebook attacks are therefore not an issue for larger blocks of 128 bits or more.

How to Construct Block Ciphers

There exist hundreds of block ciphers but only a handful of techniques to construct one. In practice, a block cipher isn't a gigantic algorithm but a

repetition of *rounds*, a short sequence of operations that's weak on its own but strong in number. There are two main techniques to construct a round: substitution–permutation networks (as in AES) and Feistel schemes (as in DES). In this section, you'll look at these techniques after viewing an attack that works when all rounds are identical to each other.

A Block Cipher's Rounds

Computing a block cipher boils down to computing a sequence of rounds. In a block cipher, a round is a basic transformation that's simple to specify and to implement and is iterated several times to form the block cipher's algorithm. This construction, consisting of a small component repeated many times, is simpler to implement and to analyze than a construction that consists of a single huge algorithm.

For example, a block cipher with three rounds encrypts a plaintext by computing $C = \mathbf{R}_3(\mathbf{R}_2(\mathbf{R}_1(P)))$, where the rounds are $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$, and P is a plaintext. Each round should also have an inverse so it's possible for a recipient to compute back to plaintext. Specifically, $P = \mathbf{iR}_1(\mathbf{iR}_2(\mathbf{iR}_3(C)))$, where $\mathbf{iR}_1$ is the inverse of $\mathbf{R}_1$, and so on.

The round functions—$\mathbf{R}_1$, $\mathbf{R}_2$, and so forth—are usually identical algorithms, but they are parameterized by a value we call the *round key*. Two round functions with two distinct round keys will behave differently and will therefore produce distinct outputs if fed with the same input.

We derive round keys from the main key, K, using a *key schedule* algorithm. For example, $\mathbf{R}_1$ takes the round key K_1, $\mathbf{R}_2$ takes the round key K_2, and so on.

Round keys should be different from each other in every round. For that matter, not all round keys should be equal to the key K; otherwise, all the rounds would be identical, and the block cipher would be less secure, as I'll describe next.

The Slide Attack and Round Keys

In a block cipher, no round should be identical to another round in order to avoid a *slide attack*. As Figure 4-1 shows, slide attacks look for two plaintext/ciphertext pairs (P_1, C_1) and (P_2, C_2), where $P_2 = \mathbf{R}(P_1)$ if $\mathbf{R}$ is the cipher's round. When rounds are identical, the relation between the two plaintexts, $P_2 = \mathbf{R}(P_1)$, implies the relation $C_2 = \mathbf{R}(C_1)$ between their respective ciphertexts. Figure 4-1 shows three rounds, but the relation $C_2 = \mathbf{R}(C_1)$ will hold no matter the number of rounds, be it 3, 10, or 100. The problem is that knowing the input and output of a single round often helps recover the key. (For details, read the 1999 paper "Advanced Slide Attacks" by Alex Biryukov and David Wagner, available at *https://www.iacr.org/archive/eurocrypt2000/1807/18070595-new.pdf*.)

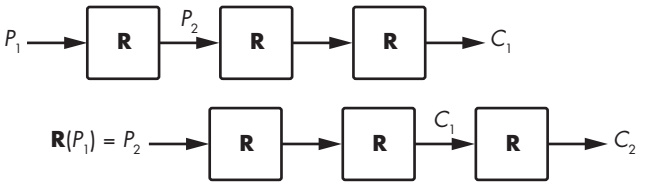


Figure 4-1: The principle of the slide attack against block ciphers with identical rounds

Using different round keys as parameters ensures that the rounds will behave differently and thus foil slide attacks.

NOTE *One potential byproduct and benefit of using round keys is protection against* side-channel attacks*, or attacks that exploit information leaked from the implementation of a cipher (for example, electromagnetic emanations). If the transformation from the main key,* K*, to a round key,* K_i*, isn't invertible, then if an attacker finds* K_i*, they can't use that key to find* K*. Unfortunately, few block ciphers have a one-way key schedule. The key schedule of AES allows attackers to compute* K *from any round key,* K_i*, for example.*

Substitution–Permutation Networks

If you've read textbooks about cryptography, you've likely come across confusion and diffusion. *Confusion* means that the input (plaintext and encryption key) undergoes complex transformations, and *diffusion* means that these transformations depend equally on all bits of the input. At a high level, confusion is about depth, whereas diffusion is about breadth. In the design of a block cipher, confusion and diffusion take the form of substitution and permutation operations, which we combine within substitution–permutation networks (SPNs).

Substitution often appears in the form of *S-boxes*, or *substitution boxes*, which are small lookup tables that transform chunks of 4 or 8 bits. For example, the first of the eight S-boxes of the block cipher Serpent is composed of the 16 elements (3 8 f 1 a 6 5 b e d 4 2 7 0 9 c), where each element represents a 4-bit nibble. This particular S-box maps the 4-bit nibble 0000 to 3 (0011), the 4-bit nibble 0101 (5 in decimal) to 6 (0110), and so on.

NOTE *S-boxes must be carefully chosen to be cryptographically strong: they should be as nonlinear as possible (inputs and outputs should be related with complex equations) and have no statistical bias (meaning, for example, that flipping an input bit should potentially affect any of the output bits).*

The permutation in a substitution–permutation network can be as simple as changing the order of the bits, which is easy to implement but doesn't mix up the bits very much. Instead of a reordering of the bits, some ciphers use basic linear algebra and matrix multiplications to mix up the bits: they perform a series of multiplication operations with fixed values (the matrix's

coefficients) and then add the results. Such operations can quickly create dependencies between all the bits within a cipher and thus ensure strong diffusion. For example, the block cipher FOX transforms a 4-byte vector (a, b, c, d) to (a', b', c', d'), which we define as follows:

$$
\begin{aligned}
a' &= a + b + c + (2 \times d) \\
b' &= a + (253 \times b) + (2 \times c) + d \\
c' &= (253 \times a) + (2 \times b) + c + d \\
d' &= (2 \times a) + b + (253 \times c) + d
\end{aligned}
$$

In these equations, we interpret the numbers 2 and 253 as binary polynomials rather than integers; hence, we define additions and multiplications a bit differently than what we're used to. For example, instead of having $2 + 2 = 4$, we have $2 + 2 = 0$. Regardless, each byte in the initial state affects all 4 bytes in the final state.

Feistel Schemes

In the 1970s, IBM engineer Horst Feistel designed a block cipher, Lucifer, that works as follows:

1. Split the 64-bit block into two 32-bit halves, L and R.
2. Set L to $L \oplus \mathbf{F}(R)$, where $\mathbf{F}$ is a substitution–permutation round.
3. Swap the values of L and R.
4. Go to step 2 and repeat 15 times.
5. Merge L and R into the 64-bit output block.

This construction is a *Feistel scheme*, as Figure 4-2 shows. The left side is the scheme as just described; the right side is a functionally equivalent representation where, instead of swapping L and R, rounds alternate the operations $L = L \oplus \mathbf{F}(R)$ and $R = R \oplus \mathbf{F}(L)$.

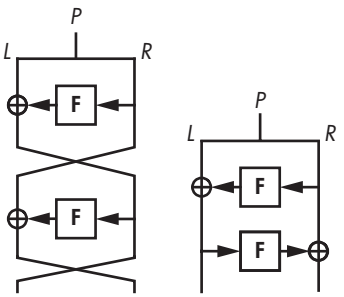


Figure 4-2: The Feistel scheme block cipher construction in two equivalent forms

I've omitted the keys from Figure 4-2 to simplify the diagrams, but note that the first $\mathbf{F}$ takes a round key, K_1, and the second $\mathbf{F}$ takes another round

key, K_2. In DES, the **F** functions take a 48-bit round key, which it derives from the 56-bit key, *K*.

In a Feistel scheme, the **F** function can be either a pseudorandom permutation (PRP) or a pseudorandom function (PRF). A PRP yields distinct outputs for any two distinct inputs, whereas a PRF will have values *X* and *Y* for which $\mathbf{F}(X) = \mathbf{F}(Y)$. But in a Feistel scheme, that difference doesn't matter as long as **F** is cryptographically strong.

How many rounds should there be in a Feistel scheme? Well, DES performs 16 rounds, whereas GOST 28147-89 performs 32 rounds. If the **F** function is as strong as possible, four rounds are sufficient in theory, but real ciphers use more rounds to defend against potential weaknesses in **F**.

The Advanced Encryption Standard

AES is the most-used cipher in the world. Prior to the adoption of AES, the standard cipher in use was DES, with its ridiculous 56-bit security, as well as the upgraded version of DES known as Triple DES, or 3DES. Although 3DES provides a higher level of security (112-bit security), it's inefficient because the key needs to be 168 bits long to get 112-bit security, and it's slow in software (DES was created to be fast in integrated circuits, not on mainstream CPUs). AES fixes both issues.

NIST standardized AES in 2000 as a replacement for DES, at which point it became the world's de facto encryption standard. Most commercial encryption products today support AES, and the NSA has approved it for protecting top-secret information. (Some countries do prefer to use their own ciphers, largely because they don't want to use a US standard, but AES is actually more Belgian than it is American.)

NOTE *AES went by the name* Rijndael *(a portmanteau for its inventors' names, Rijmen and Daemen, pronounced like "rain-dull") when it was one of the 15 candidates in the AES competition, the process held by NIST from 1997 to 2000 to specify "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century," as stated in the 1997 announcement of the competition in the* Federal Register. *The AES competition was a kind of "Got Talent" competition for cryptographers, where anyone could participate by submitting a cipher or breaking other contestants' ciphers.*

AES Internals

AES processes blocks of 128 bits using a secret key of 128, 192, or 256 bits, with the 128-bit key being the most common because it makes encryption slightly faster and because the difference between 128- and 256-bit security is meaningless for most applications.

Whereas some ciphers work with individual bits or 64-bit words, AES manipulates *bytes*. It views a 16-byte plaintext as a two-dimensional array of bytes ($s = s_0, s_1, \ldots, s_{15}$), as Figure 4-3 illustrates. (We use the letter *s*

because this array is the *internal state*, or just *state*.) AES transforms the bytes, columns, and rows of this array to produce a final value that is the ciphertext.

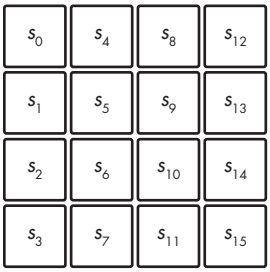


Figure 4-3: The internal state of AES as a 4×4 array of 16 bytes

To transform its state, AES uses an SPN structure as in Figure 4-4, with 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

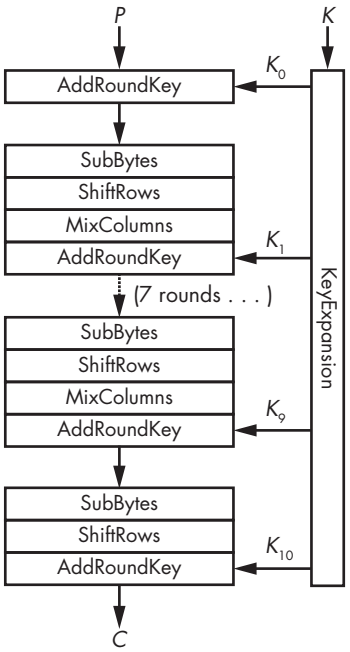


Figure 4-4: The internal operations of AES

Figure 4-4 shows the four building blocks of an AES round (note that all but the last round are a sequence of SubBytes, ShiftRows, MixColumns, and AddRoundKey):

AddRoundKey XORs a round key to the internal state.

SubBytes Replaces each byte (s_0, s_1 . . . , s_{15}) with another byte according to an S-box. In this example, the S-box is a lookup table of 256 elements.

ShiftRows Shifts the ith row of i positions, for i ranging from 0 to 3 (see Figure 4-5).

MixColumns Applies the same linear transformation to each of the four columns of the state (that is, each group of cells with the same shade of gray, as on the left side of Figure 4-5).

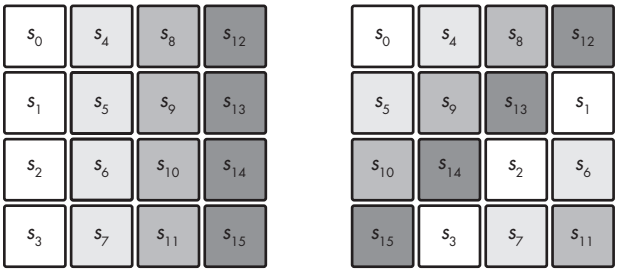


Figure 4-5: ShiftRows rotates bytes within each row of the internal state.

Remember that in an SPN, the *S* stands for substitution and the *P* for permutation. Here, the substitution layer is SubBytes, and the permutation layer is the combination of ShiftRows and MixColumns.

The key schedule function KeyExpansion, as Figure 4-4 shows, is the AES key schedule algorithm. This expansion creates 11 round keys (K_0, K_1, . . . , K_{10}) of 16 bytes each from the 16-byte key, using the same S-box as SubBytes and a combination of XORs. One important property of KeyExpansion is that given any round key, K_i, an attacker can determine all other round keys as well as the main key, K, by reversing the algorithm. The ability to get the key from any round key reduces the cipher's resistance to side-channel attacks, where an attacker may easily recover a round key.

Without these operations, AES would be totally insecure. Each operation contributes to AES's security in a specific way:

- Without KeyExpansion, all rounds would use the same key, K, and AES would be vulnerable to slide attacks.
- Without AddRoundKey, encryption wouldn't depend on the key; hence, anyone could decrypt any ciphertext without the key.
- SubBytes brings nonlinear operations, which add cryptographic strength. Without it, AES would just be a large system of linear equations that can be solved using high school algebra (namely, Gaussian elimination).
- Without ShiftRows, changes in a given column would never affect the other columns, meaning you could break AES by building four 2^{32}-element codebooks for each column. (Remember that in a

secure block cipher, flipping a bit in the input should affect all the output bits.)

- Without MixColumns, changes in a byte wouldn't affect any other bytes of the state. A chosen-plaintext attacker could then decrypt any ciphertext after storing 16 lookup tables of 256 bytes each that hold the encrypted values of each possible value of a byte.

Notice in Figure 4-4 that the last round of AES doesn't include the MixColumns operation. That operation is omitted to save useless computation: because MixColumns is linear, you could cancel its effect in the very last round by combining bits in a way that doesn't depend on their value or the key. However, we can't invert SubBytes without the state's value being known prior to AddRoundKey.

To decrypt a ciphertext, AES unwinds each operation by taking its inverse function: the inverse lookup table of SubBytes reverses the SubBytes transformation, ShiftRow shifts in the opposite direction, MixColumns's inverse is applied (as in the matrix inverse of the matrix encoding its operation), and AddRoundKey's XOR is unchanged because the inverse of an XOR is another XOR.

AES in Action

As an exercise, you can use Python's `cryptography` library to encrypt and decrypt a block of data with AES, as in Listing 4-1.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Pick a random 16-byte key using Python's crypto PRNG.
k = urandom(KEY_SIZE)
print(f"k = {k.hex()}")

# Create an instance of AES-128.
aes = Cipher(algorithms.AES(k), modes.ECB())
aes_ecb_encryptor = aes.encryptor()

# Set plaintext p to the all-zero string.
p = bytes([0x00] * BLOCK_SIZE)

# Encrypt plaintext p to ciphertext c.
c = aes_ecb_encryptor.update(p) + aes_ecb_encryptor.finalize()
print(f"enc({p.hex()}) = {c.hex()}")

# Decrypt ciphertext c to plaintext p.
aes_ecb_decryptor = aes.decryptor()
p = aes_ecb_decryptor.update(c) + aes_ecb_decryptor.finalize()
print(f"dec({c.hex()}) = {p.hex()}")
```

Listing 4-1: AES encryption and decryption of a block in Python

Running this script produces something like the following output:

```
$ ./aes_block.py
k = 2c6202f9a582668aa96d511862d8a279
enc(00000000000000000000000000000000) = 12b620bb5eddcde9a07523e59292a6d7
dec(12b620bb5eddcde9a07523e59292a6d7) = 00000000000000000000000000000000
```

You'll get different results because the key is randomized at every new execution.

How to Implement AES

Real AES software works differently than the algorithm in Figure 4-4. You won't find production-level AES code calling a `SubBytes()` function, then a `ShiftRows()` function, and then a `MixColumns()` function because that would be inefficient. Instead, fast AES software uses table-based implementations and native instructions.

Table-Based Implementations

Table-based implementations of AES replace the sequence SubBytes-ShiftRows-MixColumns with a combination of XORs and lookups in tables hardcoded into the program and loaded in memory at execution time. This is possible because MixColumns is equivalent to XORing four 32-bit values, where each depends on a single byte from the state and on SubBytes. Thus, you can build four tables with 256 entries each, one for each byte value, and implement the sequence SubBytes-MixColumns by looking up four 32-bit values and XORing them together.

For example, the table-based C implementation in the OpenSSL toolkit looks like Listing 4-2.

```
  /* Round 1: */
  t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
  t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
  t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
  t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
  /* Round 2: */
  s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
  s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
  s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
  s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
--snip--
```

Listing 4-2: An excerpt of the table-based C implementation of AES in OpenSSL

A basic table-based implementation of AES encryption needs four 4KB's worth of tables because each table stores 256 32-bit values, which occupy $256 \times 32 = 8{,}192$ bits, or 1KB. Decryption requires another four tables and thus 4KB more. But there are tricks to reduce the storage from 4KB to 1, or even less.

Alas, table-based implementations are vulnerable to *cache-timing attacks*, which exploit timing variations when a program reads or writes elements in cache memory. Access time varies depending on the relative position in cache memory of the accessed elements. Timings thus leak information about the accessed element, which in turn leaks information on the secrets involved.

Cache-timing attacks are difficult to avoid. One obvious solution would be to ditch lookup tables altogether by writing a program whose execution time doesn't depend on its inputs, but that's almost impossible to do and still retain the same speed, so chip manufacturers have opted for a radical solution: instead of relying on potentially vulnerable software, they rely on *hardware*.

Native Instructions

AES native instructions (AES-NI) solve the problem of cache-timing attacks on AES software implementations. To understand how AES-NI works, think about the way software runs on hardware: to run a program, a microprocessor translates binary code into a series of instructions that integrated circuit components execute. For example, a `MUL` assembly instruction between two 32-bit values will activate the transistors implementing a 32-bit multiplier in the microprocessor. To implement a crypto algorithm, we usually express a combination of basic operations—additions, multiplications, XORs, and so on—and the microprocessor activates its adders, multipliers, and XOR circuits in the prescribed order.

AES native instructions take this to a whole new level by providing developers with dedicated assembly instructions that compute AES. Instead of coding an AES round as a sequence of assembly instructions, when using AES-NI, you just call the instruction `AESENC`, and the chip computes the round for you. Native instructions allow you to tell the processor to run an AES round instead of requiring you to program rounds as a combination of basic operations.

A typical assembly implementation of AES using native instructions looks like Listing 4-3.

```
PXOR       %xmm5,  %xmm0
AESENC     %xmm6,  %xmm0
AESENC     %xmm7,  %xmm0
AESENC     %xmm8,  %xmm0
AESENC     %xmm9,  %xmm0
AESENC     %xmm10, %xmm0
AESENC     %xmm11, %xmm0
AESENC     %xmm12, %xmm0
AESENC     %xmm13, %xmm0
AESENC     %xmm14, %xmm0
AESENCLAST %xmm15, %xmm0
```

Listing 4-3: An implementation of AES-128 using AES native instructions

This code encrypts the 128-bit plaintext initially in the register `xmm0`, assuming that registers `xmm5` to `xmm15` hold the precomputed round keys, with each instruction writing its result into `xmm0`. The initial `PXOR` instruction XORs the first round key prior to computing the first round, and the final `AESENCLAST` instruction performs the last round slightly differently from the others (MixColumns is omitted).

NOTE *AES is about 10 times faster on platforms that implement native instructions, which, as I write this, include virtually all laptop, desktop, and server microprocessors, as well as most mobile phones and tablets. Although Intel originally proposed AES instructions in 2008, they're also available in AMD processors, and most architectures other than x86 also have equivalent instructions implementing AES in hardware. For example, the Armv8 instruction set contains the instructions* `AESSE` *(which calculates SubBytes and ShiftRows) and* `AESMS` *(which calculates MixColumns).*

On Intel's Ice Lake microarchitecture, the `AESENC` *instruction has a latency of three cycles with a reciprocal throughput of half a cycle, meaning that a call to* `AESENC` *takes three cycles to complete, and we can make two new calls to the instruction in each cycle. In fact, the internal structure of the micro-operations performing the* `AESENC` *operation means that a new instruction computation can start before the previous one has finished. What's more, the Ice Lake architecture uses a vectorized version of AES instructions, enabling several to be initiated at the same time. For more details, see the article "Making AES Great Again" by Nir Drucker, Shay Gueron, and Vlad Krasnov, available at* https://eprint.iacr.org/2018/392.

To encrypt a series of blocks one after the other, it takes $3 \times 10 = 30$ cycles to complete the 10 rounds, or $30 / 16 = 1.875$ cycles per byte. At a frequency of 2 GHz (2×10^9 cycles per second), this gives a theoretical maximum throughput of around 1GBps. If you can process blocks in parallel, then you don't need one complete `AESENC` *call before starting another. In this case, you can make two* `AESENC` *calls per cycle and obtain two results per cycle, offering a much higher theoretical throughput (up to more than 10GBps, at 2 GHz), depending on the data size and mode of operation.*

AES Security

AES is as secure as a block cipher can be, and it will never be broken. Fundamentally, AES is secure because all output bits depend on all input bits in some complex, pseudorandom way. To achieve this, the designers of AES carefully chose each component for a particular reason—MixColumns for its maximal diffusion properties and SubBytes for its optimal nonlinearity. This composition protects AES against whole classes of cryptanalytic attacks.

But there's no proof that AES is immune to all possible attacks. For one thing, we don't know what all possible attacks are, and we don't always know how to prove that a cipher is secure against a given attack. The only way to really gain confidence in the security of AES is to crowdsource attacks: have many skilled people attempt to break AES and, ideally, fail to do so.

After more than 15 years and hundreds of research publications, we've only scratched the surface of the theoretical security of AES. In 2011, cryptanalysts found a way to recover an AES-128 key by performing about

2^{126} operations instead of 2^{128}, a speedup of a factor of 4. But this "attack" requires a high number of plaintext–ciphertext pairs—about 2^{88} bits' worth. It's a nice finding but not one you need to worry about.

You should care about a million things when implementing and deploying crypto, but AES security is not one of them. The biggest threat to block ciphers isn't in their core algorithms but in their modes of operation. If you've chosen an incorrect mode or misused the right one, even a strong cipher like AES won't save you.

Modes of Operation

In Chapter 1, I explained how encryption schemes combine a permutation with a mode of operation to handle messages of any length. In this section, I'll cover the main modes of operations that block ciphers use, their security and functional properties, and how (not) to use them. I'll begin with the dumbest one: electronic codebook.

Electronic Codebook Mode

The simplest of the block cipher encryption modes is electronic codebook (ECB), which is barely a mode of operation at all. ECB takes plaintext blocks $P_1, P_2, \ldots, P_N$ and processes each independently by computing $C_1 = \mathbf{E}(K, P_1)$, $C_2 = \mathbf{E}(K, P_2)$, and so on, as Figure 4-6 shows. It's a simple operation but also an insecure one—ECB is insecure, and you shouldn't use it.

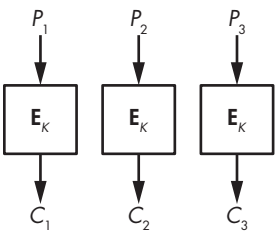


Figure 4-6: ECB mode

Marsh Ray, a cryptographer at Microsoft, once said, "Everybody knows ECB mode is bad because we can see the penguin." He was referring to a famous illustration of ECB's insecurity that uses an image of Linux's mascot, Tux, as in Figure 4-7.

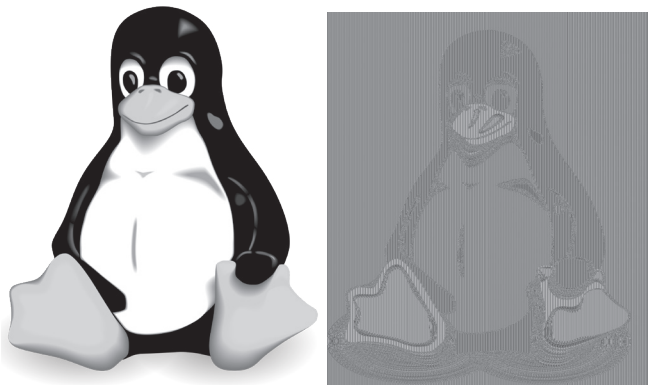


Figure 4-7: The original image (left) and the ECB-encrypted image (right)

The original image of Tux is on the left, and the ECB-encrypted image that uses AES (though the underlying cipher doesn't matter) is on the right. It's easy to see the penguin's shape in the encrypted version because ECB encrypted all the blocks of one shade of gray in the original image to the same new shade of gray in the new image; in other words, ECB encryption gives you the same image with different colors.

The Python program in Listing 4-4 also shows ECB's insecurity. It picks a pseudorandom key and encrypts a 32-byte message `p` containing two blocks of null bytes. Notice that encryption yields two identical blocks and that repeating encryption with the same key and the same plaintext yields the same two blocks again.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# The blocks() function splits a data string into space-separated blocks.
def blocks(data):
    split = [data[i:i+BLOCK_SIZE].hex() for i in range(0, len(data), BLOCK_SIZE)]
    return ' '.join(split)

k = urandom(KEY_SIZE)
print(f"k = {k.hex()}")

# Create an instance of AES-128 to encrypt and decrypt.
aes = Cipher(algorithms.AES(k), modes.ECB())
aes_ecb_encryptor = aes.encryptor()

# Set plaintext p as two blocks of zeros.
p = bytes([0x00] * 2 * BLOCK_SIZE)
```

```
# Encrypt plaintext p to ciphertext c.
c = aes_ecb_encryptor.update(p) + aes_ecb_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")
```

Listing 4-4: Using AES in ECB mode in Python

Running this script gives ciphertext blocks like this example:

```
$ ./aes_ecb.py
k = 50a0ebeff8001250e87d31d72a86e46d
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
5eb4b7af094ef7aca472bbd3cd72f1ed 5eb4b7af094ef7aca472bbd3cd72f1ed
```

When using the ECB mode, identical ciphertext blocks reveal identical plaintext blocks to an attacker, whether those are blocks within a single ciphertext or across different ciphertexts. This shows that block ciphers in ECB mode aren't semantically secure.

Another problem with ECB is that it takes only complete blocks of data, so if blocks were 16 bytes, as in AES, you could encrypt only chunks of 16 bytes, 32 bytes, 48 bytes, or any other multiple of 16 bytes. There are a few ways to deal with this, as you'll see with the next mode, CBC. (I won't tell you how these tricks work with ECB because you shouldn't use ECB in the first place.)

Cipher Block Chaining Mode

Cipher block chaining (CBC) is like ECB but with a small twist that makes a big difference: instead of encrypting the ith block, P_i, as $C_i = \mathbf{E}(K, P_i)$, CBC sets $C_i = \mathbf{E}(K, P_i \oplus C_{i-1})$, where C_{i-1} is the previous ciphertext block—thereby *chaining* the blocks C_{i-1} and C_i. When encrypting the first block, P_1, there is no previous ciphertext block to use, so CBC takes a random initial value (IV), as Figure 4-8 illustrates.

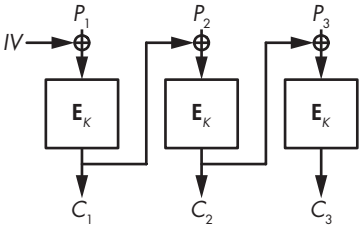


Figure 4-8: CBC mode

CBC mode makes each ciphertext block dependent on all the previous blocks and ensures that identical plaintext blocks won't be identical ciphertext blocks. The random initial value guarantees that two identical plaintexts will encrypt to distinct ciphertexts when calling the cipher twice with two distinct initial values.

Listing 4-5 illustrates these two benefits. This program takes an all-zero, 32-byte message (like the one in Listing 4-4), encrypts it twice with CBC, and shows the two ciphertexts. The bolded line `iv = urandom(BLOCK_SIZE)` picks a new random IV for each new encryption.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# The blocks() function splits a data string into space-separated blocks.
def blocks(data):
    split = [data[i:i+BLOCK_SIZE].hex() for i in range(0, len(data), BLOCK_SIZE)]
    return ' '.join(split)

# Pick a random key.
k = urandom(KEY_SIZE)
print(f"k  = {k.hex()}")

# Pick a random IV.
iv = urandom(BLOCK_SIZE)
print(f"iv = {iv.hex()}")

# Pick an instance of AES in CBC mode.
aes_cbc_encryptor = Cipher(algorithms.AES(k), modes.CBC(iv)).encryptor()

# Set plaintext p as two blocks of zeros.
p = bytes([0x00] * 2 * BLOCK_SIZE)

c = aes_cbc_encryptor.update(p) + aes_cbc_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")

# Now with a different IV and the same key
iv = urandom(BLOCK_SIZE)
print(f"iv = {iv.hex()}")

aes_cbc_encryptor = Cipher(algorithms.AES(k), modes.CBC(iv)).encryptor()
c = aes_cbc_encryptor.update(p) + aes_cbc_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")
```

Listing 4-5: Using AES in CBC mode

The two plaintexts are the same (two all-zero blocks), but the encrypted blocks should be distinct, as in this example execution:

```
$ ./aes_cbc.py
k = 9cf0d31ad2df24f3cbbefc1e6933c872
iv = 0a75c4283b4539c094fc262aff0d17af
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
370404dcab6e9ecbc3d24ca5573d2920 3b9e5d70e597db225609541f6ae9804a
```

```
iv = a6016a6698c3996be13e8739d9e793e2
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
655e1bb3e74ee8cf9ec1540afd8b2204 b59db5ac28de43b25612dfd6f031087a
```

Alas, we often use CBC with a constant IV instead of a random one, which exposes identical plaintexts and plaintexts that start with identical blocks. For example, say CBC encrypts the two-block plaintext $P_1 \| P_2$ to the two-block ciphertext $C_1 \| C_2$. If CBC encrypts $P_1 \| P_2'$ with the same IV, where P_2' is some block distinct from P_2, then the ciphertext will look like $C_1 \| C_2'$, with C_2' different from C_2 but with the same first block C_1. Thus, an attacker can guess that the first block is the same for both plaintexts, even though they see only the ciphertexts.

NOTE *In CBC mode, decryption needs to know the IV used to encrypt, so the IV is sent along with the ciphertext, in the clear.*

With CBC, decryption can be much faster than encryption due to parallelism. While encryption of a new block, P_i, needs to wait for the previous block, C_{i-1}, decryption of a block computes $P_i = \mathbf{D}(K, C_i) \oplus C_{i-1}$, where there's no need for the previous plaintext block, P_{i-1}. This means you can decrypt all blocks in parallel simultaneously, as long as you know the previous ciphertext block, which you usually will.

Message Encryption in CBC Mode

Let's circle back to the block termination issue and look at how to process a plaintext whose length is not a multiple of the block length. For example, how would you encrypt an 18-byte plaintext with AES-CBC when blocks are 16 bytes? What do you do with the 2 bytes left? You'll look at two widely used techniques to deal with this problem. The first one, padding, makes the ciphertext a bit longer than the plaintext, while the second one, *ciphertext stealing*, produces a ciphertext of the same length as the plaintext.

Message Padding

Padding is a technique that allows you to encrypt a message of any length, even one smaller than a single block. The PKCS#7 standard and RFC 5652 specify padding for block ciphers, which we use almost everywhere we use CBC.

We use padding to expand a message to fill a complete block by adding extra bytes to the plaintext. Here are the rules for padding 16-byte blocks:

- If there's 1 byte left—for example, if the plaintext is 1 byte, 17 bytes, or 33 bytes long—pad the message with 15 bytes `0f` (15 in decimal).
- If there are 2 bytes left, pad the message with 14 bytes `0e` (14 in decimal).
- If there are 3 bytes left, pad the message with 13 bytes `0d` (13 in decimal).

If there are 15 plaintext bytes and a single byte missing to fill a block, padding adds a single `01` byte. If the plaintext is already a multiple of 16, the block length, add 16 bytes `10` (16 in decimal). The trick generalizes to any block length up to 255 bytes (for larger blocks, a byte is too small to encode values greater than 255).

Decryption of a padded message works like this:

1. Decrypt all the blocks as with unpadded CBC.
2. Make sure that the last bytes of the last block conform to the padding rule: that they finish with at least one `01` byte, at least two `02` bytes, or at least three `03` bytes, and so on. If the padding isn't valid—for example, if the last bytes are `01 02 03`—the message is rejected. Otherwise, decryption strips the padding bytes and returns the plaintext bytes left.

One downside of padding is that it makes ciphertext longer by at least 1 byte and at most a block.

Ciphertext Stealing

Ciphertext stealing is another trick we use to encrypt a message whose length isn't a multiple of the block size. Ciphertext stealing is more complex and less popular than padding, but it offers a few benefits:

- Plaintexts can be of any *bit* length, not just bytes. You can, for example, encrypt a message of 131 bits.
- Ciphertexts are exactly the same length as plaintexts.
- Ciphertext stealing is not vulnerable to padding oracle attacks, powerful attacks that sometimes work against CBC with padding (as you'll see in "Padding Oracle Attacks" on page 83).

In CBC mode, ciphertext stealing extends the last incomplete plaintext block with bits from the previous ciphertext block and then encrypts the resulting block. The last, incomplete ciphertext block is made up of the first bits from the previous ciphertext block—that is, the bits that haven't been appended to the last plaintext block.

In Figure 4-9, we have three blocks, where the last block, P_3, is incomplete (represented by a zero). If P_3 is 3 bytes, we XOR it with the last 12 bits from the previous ciphertext block $\mathbf{E}(K, P_2)$ and return the encrypted result as C_2. The last ciphertext block, C_3, then consists of the first 4 bytes of $\mathbf{E}(K, P_2)$. Decryption is simply the inverse of this operation.

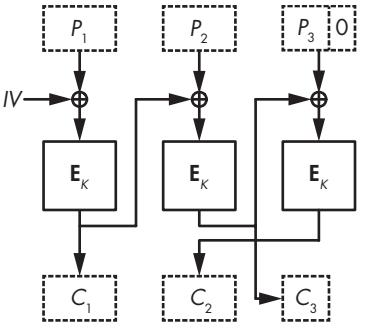


Figure 4-9: Ciphertext stealing for CBC-mode encryption

There aren't any major problems with ciphertext stealing, but it's inelegant and hard to get right, especially when NIST's standard specifies three different ways to implement it (see Special Publication 800-38A).

Counter Mode

To avoid the troubles and retain the benefits of ciphertext stealing, use counter mode (CTR). CTR is hardly a block cipher mode: it turns a block cipher into a stream cipher that just takes bits in and spits bits out and doesn't embarrass itself with the notion of blocks. (I'll discuss stream ciphers in detail in Chapter 5.)

In CTR mode (see Figure 4-10), the block cipher algorithm won't transform plaintext data. Instead, it encrypts blocks composed of a counter and a nonce. A *counter* is an integer that is incremented for each block. No two blocks should use the same counter within a message, but different messages can use the same sequence of counter values (1, 2, 3, . . .). A *nonce* is a number we use only once. It's the same for all blocks in a single message, but no two messages should use the same nonce.

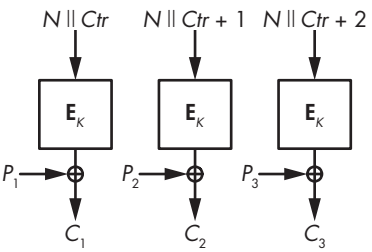


Figure 4-10: CTR mode

Figure 4-10 shows that in CTR mode, encryption XORs the plaintext and the stream taken from "encrypting" the nonce, *N*, and counter, *Ctr*. Decryption is the same, so you need only the encryption algorithm for both encryption and decryption. The Python script in Listing 4-6 gives you a hands-on example.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Pick a random key.
k = urandom(KEY_SIZE)
print(f"k  = {k.hex()}")

# And a random nonce
# (careful with random nonces, see discussion below).
n = urandom(BLOCK_SIZE)
print(f"nonce  = {n.hex()}")

# Create a 7-byte plaintext p.
p = bytes([0x00] * 7)

# Encrypt the plaintext p with AES-CTR.
aes_ctr_encryptor = Cipher(algorithms.AES(k), modes.CTR(n)).encryptor()

c = aes_ctr_encryptor.update(p) + aes_ctr_encryptor.finalize()
print(f"enc({p.hex()}) = {c.hex()}")

# Decrypt the ciphertext c.
aes_ctr_decryptor = Cipher(algorithms.AES(k), modes.CTR(n)).decryptor()
p = aes_ctr_decryptor.update(c) + aes_ctr_decryptor.finalize()
print(f"dec({c.hex()}) = {p.hex()}")

# Decrypt the ciphertext c using the encryption function.
aes_ctr_encryptor = Cipher(algorithms.AES(k), modes.CTR(n)).encryptor()
p = aes_ctr_encryptor.update(c) + aes_ctr_encryptor.finalize()
print(f"enc({c.hex()}) = {p.hex()}")
```

Listing 4-6: Using AES in CTR mode

The example execution encrypts a 4-byte plaintext and gets a 4-byte ciphertext. It then decrypts that ciphertext using the encryption function:

```
$ ./aes_ctr.py
k = 130a1aa77fa58335272156421cb2a3ea
enc(00010203) = b23d284e
enc(b23d284e) = 00010203
```

As with the initial value in CBC, the encrypter supplies CTR's nonce and sends it with the ciphertext in the clear. But unlike CBC's initial value, CTR's nonce doesn't need to be random; it simply needs to be unique. A nonce should be unique for the same reason that we shouldn't reuse a one-time pad: when calling the pseudorandom stream, S, if you encrypt P_1 to $C_1 = P_1 \oplus S$ and P_2 to $C_2 = P_2 \oplus S$ using the same nonce, then $C_1 \oplus C_2$ reveals $P_1 \oplus P_2$.

A random nonce will do the trick only if it's long enough; for example, if the nonce is n bits, chances are that after $2^{n/2}$ encryptions and as many nonces, you'll run into duplicates. Sixty-four bits are insufficient for a random nonce, since you can expect a repetition after approximately 2^{32} nonces, which is an unacceptably low number.

The counter is guaranteed unique if it's incremented for every new plaintext, and if it's long enough—for example, a 64-bit counter.

One particular benefit to CTR is that it can be faster than any other mode. Not only is it parallelizable, but you can also start encrypting even before knowing the message by picking a nonce and computing the stream that you'll later XOR with the plaintext.

NOTE *Depending on the CTR version we implement, we may either concatenate the nonce the API uses as an argument to a counter (as in Figure 4-10) or directly consider a counter as wide as a block.*

How Things Can Go Wrong

There are two must-know attacks on block ciphers: meet-in-the-middle attacks, a technique discovered in the 1970s but still used in many cryptanalytic attacks (not to be confused with man-in-the-middle attacks), and padding oracle attacks, a class of attacks discovered in 2002 by academic cryptographers, then mostly ignored, and finally rediscovered a decade later along with several vulnerable applications.

Meet-in-the-Middle Attacks

The 3DES block cipher is an upgraded version of the 1970s standard DES that takes a key of $56 \times 3 = 168$ bits (an improvement on DES's 56-bit key). But the security level of 3DES is 112 bits instead of 168 bits because of the *meet-in-the-middle (MitM)* attack.

Figure 4-11 shows that 3DES encrypts a block using the DES encryption and decryption functions: first encryption with a key, K_1; then decryption with a key, K_2; and finally encryption with another key, K_3. If $K_1 = K_2$, the first two calls cancel themselves out, and 3DES boils down to a single DES with key K_3. 3DES does encrypt-decrypt-encrypt rather than encrypting thrice to allow systems to emulate DES when necessary using the new 3DES interface.

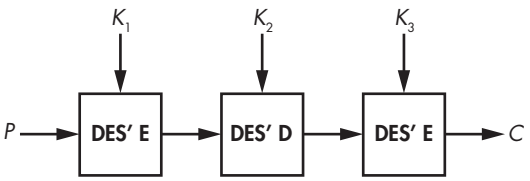


Figure 4-11: The 3DES block cipher construction

Why use triple DES and not just double DES—that is, why encrypt a plaintext P to $\mathbf{E}(K_2, \mathbf{E}(K_1, P))$? It turns out that the MitM attack makes double DES only as secure as single DES. Figure 4-12 shows the MitM attack in action.

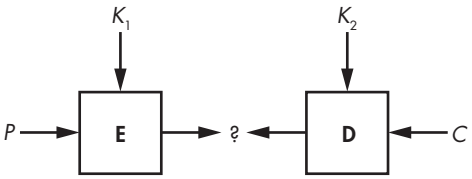


Figure 4-12: The MitM attack

The MitM attack works as follows to attack double DES:

1. Say you have P and $C = \mathbf{E}(K_2, \mathbf{E}(K_1, P))$ with two unknown 56-bit keys, K_1 and K_2. (DES takes 56-bit keys, so double DES takes 112 key bits in total.) You build a key-value table with 2^{56} entries of $\mathbf{E}(K_1, P)$, where $\mathbf{E}$ is the DES encryption function and K_1 is the value stored.
2. For all 2^{56} values of K_2, compute $\mathbf{D}(K_2, C)$ and check whether the resulting value appears in the table as an index (thus as a middle value, represented by a question mark in Figure 4-12).
3. If you find a middle value as an index of the table, fetch the corresponding K_1 from the table and verify that the (K_1, K_2) found is the right one by using other pairs of P and C. Encrypt P using K_1 and K_2 and then check that the ciphertext obtained is the given C.

This method recovers K_1 and K_2 by performing about 2^{57} instead of 2^{112} operations: step 1 encrypts 2^{56} blocks, and then step 2 decrypts at most 2^{56} blocks, for $2^{56} + 2^{56} = 2^{57}$ operations in total. You also need to store 2^{56} elements of 15 bytes each, or about 1 exabyte. That's a lot, but there's a trick that allows you to run the same attack with only negligible memory (as you'll see in Chapter 6).

You can apply the MitM attack to 3DES in almost the same way you would to double DES, except that the third stage will go through all 2^{112} values of K_2 and K_3. The whole attack thus succeeds after performing about 2^{112} operations, meaning that 3DES gets only 112-bit security despite having 168 bits of key material.

Padding Oracle Attacks

We conclude this chapter with one of the simplest and yet most devastating attacks of the 2000s: the padding oracle attack. Remember that padding fills the plaintext with extra bytes to fill a block. A plaintext of 111 bytes, for example, is a sequence of six 16-byte blocks followed by 15 bytes. In this case, forming a complete block padding adds a `01` byte. For a 110-byte plaintext, padding adds 2 `02` bytes. For a 109-byte plaintext, it adds 3 `03` bytes, and so on, up to the case where we add 16 `10` bytes, where the hexadecimal value `10` is equal to 16.

A *padding oracle* is a system that behaves differently depending on whether the padding in a CBC-encrypted ciphertext is valid. You can see it as a black box or an API that returns either a *success* or an *error* value. You can, for example, obtain a padding oracle in a service on a remote host that sends error messages when it receives malformed ciphertexts. Given such an oracle, padding oracle attacks record which inputs have a valid padding and which don't and then exploit this information to decrypt chosen ciphertext values.

Say you want to decrypt a ciphertext block C_2. I'll call X the value you're looking for, namely, $\mathbf{D}(K, C_2)$, and P_2 the block obtained after decrypting in CBC mode (see Figure 4-13). If you pick a random block C_1 and send the two-block ciphertext $C_1 \,||\, C_2$ to the oracle, decryption will succeed only if $C_1 \oplus X = P_2$ ends with valid padding—a single `01` byte, two `02` bytes, or three `03` bytes, and so on.

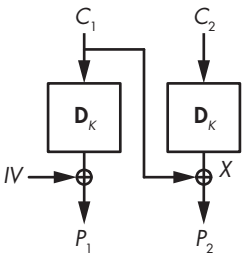


Figure 4-13: Padding oracle attacks recover X by choosing C_1 and checking the validity of padding.

Based on this observation, padding oracle attacks on CBC encryption can decrypt a block C_2 like this (bytes are denoted in array notation: $C_1[0]$ is C_1's first byte, $C_1[1]$ its second byte, and so on up to $C_1[15]$, C_1's last byte):

1. Pick a random block C_1 and vary its last byte until the padding oracle accepts the ciphertext as valid. Usually, in a valid ciphertext, $C_1[15] \oplus X[15] = \texttt{01}$, so you'll find $X[15]$ after trying around 128 values of $C_1[15]$.
2. Find the value $X[14]$ by setting $C_1[15]$ to $X[15] \oplus \texttt{02}$ and searching for the $C_1[14]$ that gives correct padding. When the oracle accepts the ciphertext as valid, it means you've found $C_1[14]$ such that $C_1[14] \oplus X[14] = \texttt{02}$.
3. Repeat steps 1 and 2 for all 16 bytes.

The attack needs on average 128 queries to the oracle for each of the 16 bytes, which is about 2,000 queries in total. (Note that each query must use the same initial value.)

NOTE *In practice, implementing a padding oracle attack is a bit more complicated than what I've described because you have to deal with wrong guesses at step 1. A ciphertext may have valid padding not because* P_2 *ends with a single* `01` *but because it ends with two* `02` *bytes or three* `03` *bytes. You can manage this by testing the validity of ciphertexts where more bytes are modified.*

Further Reading

There's a lot to say about block ciphers, be it in how algorithms work or in how they can be attacked. For instance, Feistel networks and SPNs aren't the only ways to build a block cipher. The block ciphers IDEA and FOX use the Lai–Massey construction, and Threefish uses ARX networks, a combination of addition, word rotations, and XORs.

There are also many more modes than ECB, CBC, and CTR. Some modes are folklore techniques that nobody uses, like CFB and OFB, while others are for specific applications, like XTS for tweakable encryption or GCM for authenticated encryption.

I've discussed Rijndael, the AES winner, but there were 14 other algorithms in the race: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, Magenta, MARS, RC6, SAFER+, Serpent, and Twofish. I recommend looking them up to see how they work, how they were designed, how they have been attacked, and how fast they are. It's also worth checking out the NSA's designs (Skipjack and, more recently, SIMON and SPECK) and more recent "lightweight" block ciphers such as GIFT, KATAN, PRESENT, or PRINCE.

5

STREAM CIPHERS

Symmetric ciphers can be either block ciphers or stream ciphers. Recall from Chapter 4 that block ciphers mix chunks of plaintext bits together with key bits to produce chunks of ciphertext of the same size, usually 64 or 128 bits. Stream ciphers, on the other hand, don't mix plaintext and key bits; instead, they generate pseudorandom bits from the key and encrypt the plaintext by XORing it with the pseudorandom bits, in the same fashion as the one-time pad explained in Chapter 1.

Stream ciphers are sometimes shunned because they have historically been more fragile than block ciphers and are more often broken—both the experimental ones designed by amateurs and the ciphers deployed in systems used by millions, including mobile phones, Wi-Fi, and public transport smart cards. But fortunately, although it's taken almost 20 years, we now

know how to design secure stream ciphers, and we trust them to protect Bluetooth connections, mobile 4G communications, and TLS connections.

This chapter first presents how stream ciphers work and discusses the two main classes of stream ciphers: stateful and counter based. We'll then study hardware- and software-oriented stream ciphers and look at some insecure ciphers (A5/1 as used in GSM mobile communications, and RC4 in older version of TLS) and secure, state-of-the-art ones (Grain-128a for hardware and Salsa20 for software).

How Stream Ciphers Work

Stream ciphers are more akin to deterministic random bit generators (DRBGs) than block ciphers because they generate a stream of pseudorandom bits rather than directly mixing plaintext data.

What sets stream ciphers apart from DRBGs is that DRBGs take a single input value, whereas stream ciphers take two values: a key and a nonce. The key should be secret and is usually 128 or 256 bits. The nonce doesn't have to be secret, but it should be unique for each key and is usually between 64 and 128 bits.

Stream ciphers produce a pseudorandom stream of bits we call the *keystream*. To encrypt the keystream, we XOR it to a plaintext and then XOR it again to the ciphertext to decrypt it. Figure 5-1 shows the basic stream cipher encryption operation, where **SC** is the stream cipher algorithm, *KS* the keystream, *P* the plaintext, and *C* the ciphertext.

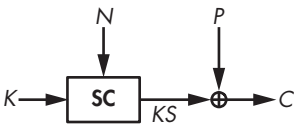


Figure 5-1: How stream ciphers encrypt, taking a secret key, K*, and a public nonce,* N

A stream cipher computes $KS = \mathbf{SC}(K, N)$, encrypts as $C = P \oplus KS$, and decrypts as $P = C \oplus KS$. The encryption and decryption functions are the same because both do the same thing—namely, XOR bits with the keystream. That's why, for example, certain cryptographic libraries provide a single `encrypt` function for both encryption and decryption.

Stream ciphers allow you to encrypt a message with key K_1 and nonce N_1 and then encrypt another message with key K_1 and nonce N_2 that's different from N_1, or with key K_2, which is different from K_1 and nonce N_1. However, you should never again encrypt with K_1 and N_1 because you would use the same keystream *KS* twice. That is, you'd have a first ciphertext $C_1 = P_1 \oplus KS$ and a second ciphertext $C_2 = P_2 \oplus KS$, and if you know P_1, then you could determine $P_2 = C_1 \oplus C_2 \oplus P_1$.

NOTE Nonce *is short for "number used only once." In the context of stream ciphers, we sometimes call it the* IV*, for "initial value."*

From a high-level perspective, there are two types of stream ciphers: stateful and counter based. *Stateful stream ciphers* have a secret internal state that evolves throughout keystream generation. The cipher initializes the state from the key and the nonce and then calls an update function to update the state value and produce one or more keystream bits from the state, as Figure 5-2 shows. For example, RC4 is stateful while Salsa20 is counter based.

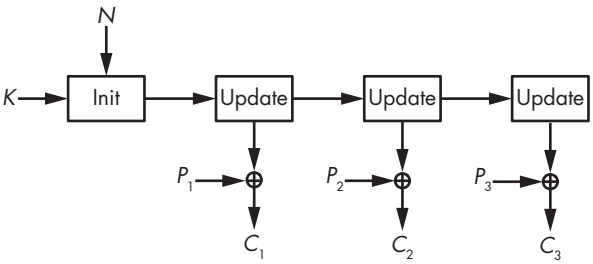


Figure 5-2: A stateful stream cipher

Counter-based stream ciphers produce chunks of a keystream from a key, a nonce, and a counter value, as in Figure 5-3. Unlike stateful stream ciphers, counter-based stream ciphers such as Salsa20 don't keep track of a secret during keystream generation, apart from the counter's value.

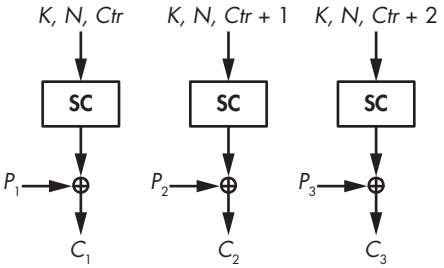


Figure 5-3: A counter-based stream cipher

These two approaches define the high-level architecture of the stream cipher, regardless of how the core algorithms work. The internals of the stream cipher also fall into two categories, depending on the target platform of the cipher: hardware oriented and software oriented.

Hardware-Oriented Stream Ciphers

When cryptographers talk about hardware, they mean application-specific integrated circuits (ASICs), programmable logic devices (PLDs), and field-programmable gate arrays (FPGAs). A cipher's hardware implementation

is an electronic circuit that implements the cryptographic algorithm at the bit level and that can't be used for anything else; in other words, the circuit is *dedicated hardware*. On the other hand, software implementations of cryptographic algorithms simply tell a microprocessor what instructions to execute in order to run the algorithm. These instructions operate on bytes or words and then call pieces of electronic circuits that implement general-purpose operations such as addition and multiplication. Software deals with bytes or words of 32 or 64 bits, whereas hardware deals with bits. The first stream ciphers worked with bits to save complex wordwise operations and thus be more efficient in hardware, their target platform at the time.

Stream ciphers were mainly used for hardware implementations because they were cheaper than block ciphers. They needed less memory and fewer logical gates than block ciphers and therefore occupied a smaller area on an integrated circuit, which reduced fabrication costs. For example, counting in gate-equivalents, the standard area metric for integrated circuits, you could find stream ciphers taking less than 1,000 gate-equivalents; by contrast, typical software-oriented block ciphers needed at least 10,000 gate-equivalents, making crypto an order of magnitude more expensive than with stream ciphers.

Today, however, block ciphers are no longer more expensive than stream ciphers—first, because there are now hardware-friendly block ciphers about as small as stream ciphers, and second, because the cost of hardware has plunged. Yet stream ciphers are often associated with hardware because they used to be the best option.

In the next section, I'll explain the basic mechanism behind hardware stream ciphers, called *feedback shift registers (FSRs)*. Almost all hardware stream ciphers rely on FSRs in some way, whether that's the A5/1 cipher used in 2G mobile phones or the more recent Grain-128a cipher.

NOTE *The first standard block cipher, the Data Encryption Standard (DES), was optimized for hardware rather than software. When the US government standardized DES in the 1970s, most target applications were hardware implementations. It's therefore no surprise that the S-boxes in DES are small and fast to compute when implemented as a logical circuit in hardware but inefficient in software. Unlike DES, the current Advanced Encryption Standard (AES) deals with bytes and is therefore more efficient in software than DES.*

Feedback Shift Registers

Countless stream ciphers use FSRs because they're simple and well understood. An FSR is an array of bits equipped with an update *feedback function*, which I'll denote as **f**. The FSR's state is stored in the array, or register, and each *update* of the FSR uses the feedback function to change the state's value and to produce one output bit.

In practice, an FSR works like this: if R_0 is the initial value of the FSR, the next state, R_1, is defined as R_0 left-shifted by 1 bit, where the bit leaving the register is returned as output and where the empty position is filled with $\mathbf{f}(R_0)$.

We repeat the same rule to compute the subsequent state values R_2, R_3, and so on. That is, given R_t, the FSR's state at time t, the next state, R_{t+1}, is the following:

$$R_{t+1} = (R_t << 1) \mid \mathbf{f}(R_t)$$

In this equation, | is the logical OR operator, and << is the shift operator, as used in the C language. For example, given the 8-bit string 00001111, we have this:

$$00001111 << 1 = 00011110$$
$$00011110 << 1 = 00111100$$
$$00111100 << 1 = 01111000$$

The bit shift moves the bits to the left, losing the leftmost bit to retain the state's bit length and zeroing the rightmost bit. The update operation of an FSR is identical, except that instead of being set to 0, the rightmost bit is set to $\mathbf{f}(R_t)$.

Consider, for example, a 4-bit FSR whose feedback function **f** XORs all 4 bits together. Initialize the state to the following:

$$1\ 1\ 0\ 0$$

Now shift the bits to the left, where 1 is output and the rightmost bit is set to the following:

$$\mathbf{f}(1100) = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Now the state becomes this:

$$1\ 0\ 0\ 0$$

The next update outputs 1, left-shifts the state, and sets the rightmost bit to the following:

$$\mathbf{f}(1000) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

Now the state is this:

$$0\ 0\ 0\ 1$$

The next three updates return three 0 bits and give the following state values:

$$0\ 0\ 1\ 1$$
$$0\ 1\ 1\ 0$$
$$1\ 1\ 0\ 0$$

We thus return to our initial state of 1100 after five iterations; updating the state five times from any of the values observed throughout this cycle

returns us to this initial value. We say that 5 is the *period* of the FSR given any one of the values 1100, 1000, 0001, 0011, or 0110. Because the period of this FSR is 5, clocking the register 10 times yields twice the same 5-bit sequence. Likewise, if you clock the register 20 times, starting from 1100, the output bits will be 11000110001100011000, or four times the same 5-bit sequence of 11000. Intuitively, such repeating patterns should be avoided, and a longer period is better for security.

NOTE *If you plan to use an FSR in a stream cipher, avoid one with short periods, which make the output more predictable. With some types of FSRs, it's easy to figure out their period, but it's almost impossible to do so with others.*

Figure 5-4 shows the structure of this cycle, along with the other cycles of that FSR, with each cycle being a circle whose dots represent a state of the register.

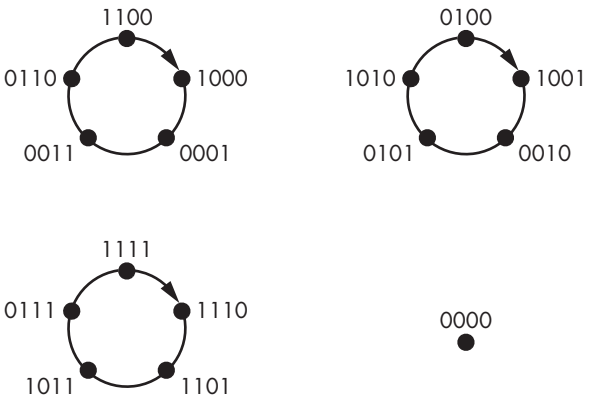


Figure 5-4: Cycles of the FSR whose feedback function XORs the 4 bits together

Indeed, this particular FSR has two other period-5 cycles—{0100, 1001, 0010, 0101, 1010} and {1111, 1110, 1101, 1011, 0111}. Note that any given state can belong to only one cycle of states. Here, we have three cycles of five states each, covering 15 of the $2^4 = 16$ possible values of our 4-bit register. The 16th possible value is 0000, which, as Figure 5-4 shows, is a period-1 cycle because the FSR transforms 0000 to 0000.

An FSR is essentially a register of bits, where each update of the register outputs a bit (the leftmost bit of the register) and where a function computes the new rightmost bit of the register. (All other bits are left-shifted.) The period of an FSR, from some initial state, is the number of updates needed until the FSR enters the same state again. If it takes N updates to do so, the FSR will produce the same N bits again and again.

Linear Feedback Shift Registers

Linear feedback shift registers (LFSRs) are FSRs with a *linear* feedback function—namely, a function that's the XOR of some bits of the state, such as the example of a 4-bit FSR in the previous section and its feedback function returning the XOR of the register's 4 bits. Recall that in cryptography, linearity is synonymous with predictability and suggestive of a simple underlying mathematical structure. And, as you might expect, thanks to this linearity, we can analyze LFSRs using notions like linear complexity, finite fields, and primitive polynomials—but I'll skip the math details and just give you the essential facts.

NOTE *In linear algebra, we define a linear transform* f *as a function that satisfies* $f(u + v) = f(u) + f(v)$*. If you know* $f(u)$ *and* $f(v)$*, you can then determine* $f(u + v)$ *without knowing* u *or* v*. With a nonlinear function, it's much more complicated; you can't easily find* $f(u + v)$ *from* $f(u)$ *and* $f(v)$*.*

The choice of which bits to XOR together is crucial for the period of the LFSR and thus for its cryptographic value. The good news is that we know how to select the position of the bits to guarantee a maximal period of $2^n - 1$. Specifically, we take the indices of the bits, from 1 for the rightmost to n for the leftmost, and write the polynomial expression $1 + X + X^2 + \ldots + X^n$, where we include the term X^i only if the ith bit is one of the bits XORed in the feedback function. The period is maximal *if and only if* that polynomial is *primitive*. To be primitive, the polynomial must have the following qualities:

- The polynomial must be irreducible, meaning that it can't be factorized—that is, written as a product of smaller polynomials. For example, $X + X^3$ is not irreducible because it's equal to $(1 + X)(X + X^2)$:

$$(1 + X)\left(X + X^2\right) = X + X^2 + X^2 + X^3 = X + X^3$$

- The polynomial must satisfy certain other mathematical properties that cannot be easily explained without nontrivial mathematical notions but are easy to test.

NOTE *The maximal period of an* n*-bit LFSR is* $2^n - 1$*, not* 2^n*, because the all-zero state always loops on itself infinitely. Because the XOR of any number of zeros is zero, new bits entering the state from the feedback functions will always be zero; hence, the all-zero state is doomed to stay all zeros.*

For example, Figure 5-5 shows a 4-bit LFSR with the feedback polynomial $1 + X + X^3 + X^4$ in which we XOR the bits at positions 1, 3, and 4 together to compute the new bit set to L_1. However, this polynomial isn't primitive because we can factorize it into $(1 + X^3)(1 + X)$.

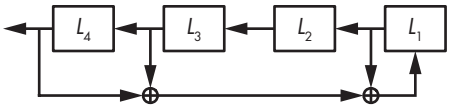


Figure 5-5: An LFSR with the feedback polynomial $1 + X + X^3 + X^4$

Indeed, the period of the LFSR in Figure 5-5 isn't maximal. To prove this, start from the state 0001:

0 0 0 1

Now left-shift by 1 bit and set the new bit to 0 + 0 + 1 = 1:

0 0 1 1

Repeating the operation five more times gives the following state values:

0 1 1 1
1 1 0 0
1 0 0 0
0 0 0 1

The state after six updates is the same as the initial one, demonstrating that we're in a period-6 cycle and proving that the LFSR's period isn't the maximal value of 15.

Let's now look at an LFSR with a maximal period, considering the LFSR in Figure 5-6.

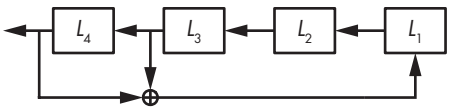


Figure 5-6: An LFSR with the feedback polynomial $1 + X^3 + X^4$, a primitive polynomial, ensuring a maximal period

This feedback polynomial is a primitive polynomial described by $1 + X^3 + X^4$, and you can verify that its period is maximal (namely, 15). From an initial value, the state evolves as follows (from 0001 to 0010, 0100, 1001, 0011, and so on):

0 0 0 1	0 0 1 1	0 1 0 1	1 1 1 0
0 0 1 0	0 1 1 0	1 0 1 1	1 1 0 0
0 1 0 0	1 1 0 1	0 1 1 1	1 0 0 0
1 0 0 1	1 0 1 0	1 1 1 1	0 0 0 1

The state spans all possible values except 0000 with no repetition until it eventually loops. This demonstrates that the period is maximal and that the feedback polynomial is primitive.

Alas, using an LFSR as a stream cipher is not secure. If n is the LFSR's bit length, an attacker needs only n output bits to recover the LFSR's initial state, allowing them to determine all previous bits and predict all future bits. This attack is possible because the LFSR is linear, implying that relations between the bits of the state obey linear equations, which are simple to solve. You can use the Berlekamp–Massey algorithm to solve the equations defined by the LFSR's mathematical structure to find not only the LFSR's initial state but also its feedback polynomial. In fact, you don't even need to know the exact length of the LFSR to succeed; you can repeat the Berlekamp–Massey algorithm for all possible values of n until you hit the right one.

The upshot is that LFSRs are cryptographically weak because they're linear. Output bits and initial state bits are related by simple and short equations that you can solve with high school linear algebra techniques. To strengthen LFSRs, let's add a pinch of nonlinearity.

Filtered LFSRs

To mitigate the insecurity of LFSRs, hide their linearity by passing their output bits through a nonlinear function before returning them to produce a *filtered LFSR*, as Figure 5-7 illustrates.

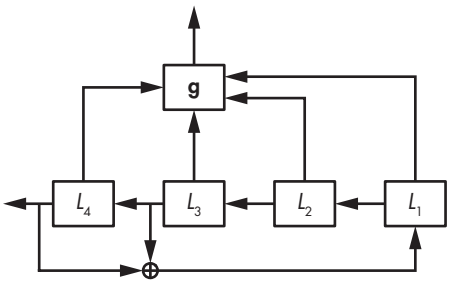


Figure 5-7: A filtered LFSR

The **g** function in Figure 5-7 must be a *nonlinear* function—one that both XORs bits together and combines them with logical AND or OR operations. For example, $L_1L_2 + L_3L_4$ is a nonlinear function (I've omitted the multiplication sign, so L_1L_2 means $L_1 \times L_2$, or L_1 & L_2 using C syntax).

NOTE *You can write feedback functions either directly in terms of an FSR's bits, like* $L_1L_2 + L_3L_4$*, or using the equivalent polynomial notation* $1 + XX^2 + X^3X^4$*. The direct notation is easier to grasp, but the polynomial notation better serves the mathematical analysis of an FSR's properties. We'll stick to the direct notation unless we care about the mathematical properties.*

Filtered LFSRs are stronger than plain LFSRs because their nonlinear function thwarts straightforward attacks. Still, more complex attacks such as the following will break the system:

Algebraic attacks Solve the nonlinear equation systems deduced from the output bits, where unknowns in the equations are bits from the LFSR state.

Cube attacks Compute derivatives of the nonlinear equations to reduce the degree of the system down to one and then solve it efficiently like a linear system.

Fast correlation attacks Exploit filtering functions that, despite their nonlinearity, tend to behave like linear functions.

The lesson here, as we've seen in previous examples, is that Band-Aids don't fix bullet holes. Patching a broken algorithm with a slightly stronger layer won't make the whole thing secure. You must fix the problem at the core.

Nonlinear FSRs

Nonlinear FSRs (NFSRs) are like LFSRs but with a nonlinear feedback function instead of a linear one. Instead of just bitwise XORs, the feedback function can include bitwise AND and OR operations—a feature with both pros and cons.

One benefit of the addition of nonlinear feedback functions is that they make NFSRs cryptographically stronger than LFSRs because the output bits depend on the initial secret state in a complex fashion, according to equations of exponential size. The LFSRs' linear function keeps the relations simple, with at most n terms ($N_1, N_2, \ldots, N_n$, if the N_is are the NFSR's state bits). For example, a 4-bit NFSR with an initial secret state (N_1, N_2, N_3, N_4) and a feedback function $N_1 + N_2 + N_1N_2 + N_3N_4$ produce a first output bit equal to the following:

$$N_1 + N_2 + N_1N_2 + N_3N_4$$

The second iteration replaces the N_1 value with that new bit. Expressing the second output bit in terms of the initial state, we get the following equation:

$$\begin{aligned}(N_1 + N_2 + N_1N_2 + N_3N_4) + N_1 + (N_1 + N_2 + N_1N_2 + N_3N_4)N_1 + N_2N_3 \\ = N_1 + N_2 + N_1N_2 + N_2N_3 + N_3N_4 + N_1N_3N_4\end{aligned}$$

This new equation has algebraic degree 3 (the highest number of bits multiplied together, here in $N_1N_3N_4$) rather than degree 2 of the feedback function, and it has six terms instead of four. As a result, iterating the nonlinear function quickly yields unmanageable equations because the size of the output grows exponentially. Although you'll never compute those equations when running the NFSR, an attacker has to solve them in order to break the system.

One downside to NFSRs is that there's no efficient way to determine an NFSR's period or to know whether its period is maximal. For an NFSR of n bits, you need to run close to 2^n trials to verify that its period is maximal. This calculation is impossible for large NFSRs of 80 bits or more.

Fortunately, there's a trick to using an NFSR without worrying about short periods: you can combine LFSRs and NFSRs to get both a guaranteed maximal period and the cryptographic strength—and that's exactly how Grain-128a works.

Grain-128a

Remember the AES competition discussed in Chapter 4, in the context of the AES block cipher? The stream cipher Grain is the offspring of a similar project called the eSTREAM competition. This competition closed in 2008 with a shortlist of recommended stream ciphers, which included four hardware-oriented ciphers and four software-oriented ones. Grain is one of these hardware ciphers, and Grain-128a is an upgraded version from the original authors of Grain. Figure 5-8 shows the action mechanism of Grain-128a.

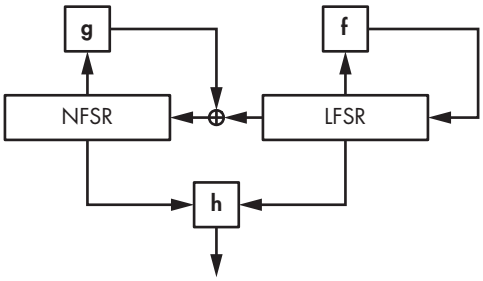


Figure 5-8: The mechanism of Grain-128a, with a 128-bit NFSR and a 128-bit LFSR

Grain-128a is about as simple as a stream cipher can be, combining a 128-bit LFSR, a 128-bit NFSR, and a filter function, **h**. The LFSR has a maximal period of $2^{128} - 1$, which ensures that the period of the whole system is at least $2^{128} - 1$ to protect against potential short cycles in the NFSR. At the same time, the NFSR and the nonlinear filter function **h** add cryptographic strength.

Grain-128a takes a 128-bit key and a 96-bit nonce. It copies the 128 key bits into the NFSR's 128 bits and copies the 96 nonce bits into the first 96 LFSR bits, filling the 32 bits left with ones and a single zero bit at the end. The initialization phase updates the whole system 256 times before returning the first keystream bit. During initialization, the bit returned by the **h** function is thus not output as a keystream but instead goes into the LFSR to ensure that its subsequent state depends on both the key and the nonce.

Grain-128a's LFSR feedback function is

$$\mathbf{f}(L) = L_{32} + L_{47} + L_{58} + L_{90} + L_{121} + L_{128}$$

where $L_1, L_2, \ldots, L_{128}$ are the bits of the LFSR. This feedback function takes only 6 bits from the 128-bit LFSR, but that's enough to get a primitive polynomial that guarantees a maximal period. The small number of bits minimizes the cost of a hardware implementation.

Here is the feedback polynomial of Grain-128a's NFSR ($N_1, \ldots, N_{128}$):

$$\mathbf{g}(N) = N_{32} + N_{37} + N_{72} + N_{102} + N_{128} + N_{44}N_{60} + N_{61}N_{125} + N_{63}N_{67} + N_{69}N_{101} + N_{80}N_{88} + N_{110}N_{111} + N_{115}N_{117} + N_{46}N_{50}N_{58} + N_{103}N_{104}N_{106} + N_{33}N_{35}N_{36}N_{40}$$

This function was carefully chosen to maximize its cryptographic strength while minimizing its implementation cost. It has an algebraic degree of 4 because its term with the most variables has four variables (namely, $N_{33}N_{35}N_{36}N_{40}$). Moreover, **g** can't be approximated by a linear function because it's highly nonlinear. Also, in addition to **g**, Grain-128a XORs the bit coming out from the LFSRs to feed the result back as the NFSR's new, rightmost bit.

The filter function **h** is another nonlinear function; it takes 9 bits from the NFSR and 7 bits from the LFSR and combines them in a way that ensures good cryptographic properties.

As I write this, there is no known attack on Grain-128a, and I'm confident that it will remain secure. Grain-128a is used in some low-end embedded systems that need a compact and fast stream cipher—typically industrial proprietary systems—which is why Grain-128a is little known in the open source software community.

A5/1

A5/1 is a stream cipher that was used to encrypt voice communications in the 2G mobile standard. The A5/1 standard was created in 1987 but only published in the late 1990s after it was reverse engineered. Attacks appeared in the early 2000s, and A5/1 was eventually broken in a way that allows actual (rather than theoretical) decryption of encrypted communications. Let's see why and how.

A5/1's Mechanism

As Figure 5-9 shows, A5/1 relies on three LFSRs and uses a trick that looks clever at first glance but actually fails to be secure.

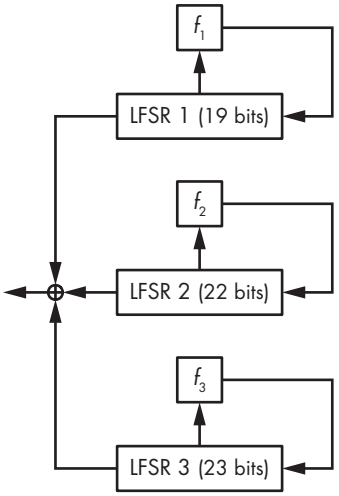


Figure 5-9: The A5/1 cipher

A5/1 uses LFSRs of 19, 22, and 23 bits, with the polynomials for each as follows:

$$1 + X^{14} + X^{17} + X^{18} + X^{19}$$
$$1 + X^{21} + X^{22}$$
$$1 + X^{8} + X^{21} + X^{22} + X^{23}$$

How could this be seen as secure with only LFSRs and no NFSR? The trick lies in A5/1's update mechanism. Instead of updating all three LFSRs at each clock cycle, the designers of A5/1 added a clocking rule that does the following:

1. Checks the value of the 9th bit of LFSR 1, the 11th bit of LFSR 2, and the 11th bit of LFSR 3, called the *clocking bits*. Of those 3 bits, either all have the same value (1 or 0) or two have the same value.
2. Clocks the registers whose clocking bits are equal to the majority value, 0 or 1. Either two or three LFSRs are clocked at each update.

Without this rule, A5/1 would provide no security whatsoever, and bypassing this rule is enough to break the cipher. However, that's easier said than done, as you'll see.

NOTE *In A5/1's irregular clocking rule, each register is clocked with a probability of 3/4 at any update. Namely, the probability that at least one other register has the same bit value is $1 - (1/2)^2$, where $(1/2)^2$ is the chance that both of the other two registers have a different bit value.*

2G communications use A5/1 with a key of 64 bits and a 22-bit nonce, which changes for every new data frame. The initialization mechanism of

A5/1 first sets all registers to zero and injects the key followed by the nonce bit by bit to each register, and after each bit is injected, the registers are updated. The system is then updated 100 times following the previously described irregular rule.

Attacks on A5/1 recover the 64-bit initial state of the system (the 19 + 22 + 23 LFSR initial value), in turn revealing the nonce (if it wasn't already known) and the key, by unwinding the initialization mechanism. The attacks are *known-plaintext attacks (KPAs)* because part of the encrypted data is known, which allows attackers to determine the corresponding keystream parts by XORing the ciphertext with the known plaintext chunks.

There are two main types of attacks on A5/1:

Subtle attacks Exploit the internal linearity of A5/1 and its simple irregular clocking system.

Brutal attacks Exploit only the short key of A5/1 and the invertibility of the frame number injection.

Let's see how these attacks work.

Subtle Attacks

We'll examine the *guess-and-determine* subtle attack. In this kind of attack, an attacker guesses certain secret values of the state to determine others. In cryptanalysis, "guessing" means brute-forcing: for each possible value of LFSRs 1 and 2 and all possible values of LFSR 3's clocking bit during the first 11 clocks, the attack reconstructs LFSR 3's bits by solving equations that depend on the bits guessed. When the guess is correct, the attacker gets the right value for LFSR 3.

The attack's pseudocode looks like this:

```
For all 2^19 values of LFSR 1's initial state
    For all 2^22 values of LFSR 2's initial state
        For all 2^11 values of LFSR 3's clocking bit during the first 11 clocks
            Reconstruct LFSR 3's initial state
            Test whether guess is correct; if yes, return; else continue
```

How efficient is this attack compared to the 2^{64}-trial brute-force search discussed in Chapter 3? This attack makes at most $2^{19} \times 2^{22} \times 2^{11} = 2^{52}$ operations in the worst case, when the algorithm succeeds only at the very last test. That's 2^{12} (or about 4,000) times faster than in the brute-force search, assuming that the last two operations in the previous pseudocode require about as much computation as testing a 64-bit key in a brute-force search. But is this assumption correct?

Recall our discussion of the full attack cost in Chapter 3. When evaluating the cost of an attack, we need to consider not only the amount of computation required to perform the attack but also parallelism and memory consumption. Neither is an issue here: as with any brute-force attack, the guess-and-determine attack is embarrassingly parallel (or N times faster when run on N cores) and doesn't need more memory than just running the cipher itself.

Our 2^{52} attack cost estimate is inaccurate for another reason. In fact, each of the 2^{52} operations (testing a key candidate) takes about four times as many clock cycles as does testing a key in a brute-force attack. The upshot is that the real cost of this particular attack is closer to $4 \times 2^{52} = 2^{54}$ operations when compared to a brute-force attack.

The guess-and-determine attack on A5/1 can decrypt encrypted mobile communications, but it takes a couple of hours to recover the key when run on a cluster of dedicated hardware devices. In other words, it's nowhere near real-time decryption. For that, we have another type of attack.

Brutal Attacks

The *time-memory trade-off (TMTO)* attack is the brutal attack on A5/1. This attack doesn't care about A5/1's internals; it cares only that its state is 64 bits long. The TMTO attack sees A5/1 as a black box that takes in a 64-bit value (the state) and spits out a 64-bit value (the first 64 keystream bits).

The idea behind the attack is to reduce the cost of a brute-force search in exchange for using a lot of memory. The simplest type of TMTO is a type of codebook attack, wherein you precompute a table of 2^{64} elements containing a combination of key and value pairs (`key:value`) and store the output value for each of the 2^{64} possible keys. To use this precomputed table for the attack, simply collect the output of an A5/1 instance and then look up in the table which key corresponds to that output. The attack itself is fast—taking only the amount of time necessary to look up a value in memory—but the creation of the table takes 2^{64} computations of A5/1. Worse, codebook attacks require an insane amount of memory: $2^{64} \times (64 + 64)$ bits, which is 2^{68} bytes or 256 exabytes. That's dozens of data centers, so we can forget about it.

TMTO attacks reduce the memory requirements of codebook attacks at the price of increased computation during the online phase of the attack. The smaller the table, the more computations required to crack a key. Regardless, it costs about 2^{64} operations to prepare the table, but that needs to be done only once.

In 2010, researchers took about two months to generate two terabytes' worth of tables, using graphics processing units (GPUs) and running 100,000 instances of A5/1 in parallel. With the help of such large tables, calls encrypted with A5/1 could be decrypted almost in real time. Telecommunication operators have implemented workarounds to mitigate the attack, but a real solution came with the 3G and 4G mobile telephony standards, which ditched A5/1 altogether.

Software-Oriented Stream Ciphers

Software stream ciphers work with bytes or 32- or 64-bit words instead of individual bits, which proves to be more efficient on modern CPUs where instructions can perform arithmetic operations on a word in the same amount of time as on a bit. Software stream ciphers are therefore better suited than

hardware ciphers for servers or browsers running on personal computers, where powerful general-purpose processors run the cipher as native software.

Today, there is considerable interest in software stream ciphers for a few reasons. First, because many devices embed powerful CPUs and hardware has become cheaper, there's less of a need for small bit-oriented ciphers. For example, the two stream ciphers in the mobile communications standard 4G (the European SNOW3G and the Chinese ZUC) work with 32-bit words and not bits, unlike the older A5/1.

Second, stream ciphers have gained popularity in software at the expense of block ciphers, notably following the fiasco of the padding oracle attack against block ciphers in CBC mode. In addition, stream ciphers are easier to specify and to implement than block ciphers: instead of mixing message and key bits together, stream ciphers just ingest key bits as a secret. In fact, one of the most popular stream ciphers is actually a block cipher in disguise: AES in counter mode (CTR).

One software stream cipher design, used by SNOW3G and ZUC, copies hardware ciphers and their FSRs, replacing bits with bytes or words. But these aren't the most interesting designs for a cryptographer. As of this writing, the two designs of most interest are RC4 and Salsa20, which are used in numerous systems, despite the fact that one is completely broken.

RC4

Designed in 1987 by Ron Rivest of RSA Security and then reverse engineered and leaked in 1994, RC4 has long been the most widely used stream cipher. RC4 has been used in countless applications, most famously in the first Wi-Fi encryption standard Wired Equivalent Privacy (WEP) and in the Transport Layer Security (TLS) protocol used to establish HTTPS connections. Unfortunately, RC4 isn't secure enough for most applications, including WEP and TLS. To understand why, let's see how RC4 works.

How RC4 Works

RC4 is among the simplest ciphers ever created. It doesn't perform any crypto-like operations, and it has no XORs, no multiplications, no S-boxes . . . nada. It simply swaps bytes. RC4's internal state is an array, S, of 256 bytes, first set to $S[0] = 0$, $S[1] = 1$, $S[2] = 2$, . . . , $S[255] = 255$, and then initialized from an n-byte K using its *key scheduling algorithm (KSA)*, which works as in the Python code in Listing 5-1.

```
j = 0
# Set S to the array S[0] = 0, S[1] = 1, ..., S[255] = 255.
S = range(256)
# Iterate over i from 0 to 255.
for i in range(256):
    # Compute the sum of v.
    j = (j + S[i] + K[i % n]) % 256
    # Swap S[i] and S[j].
    S[i], S[j] = S[j], S[i]
```

Listing 5-1: The key scheduling algorithm of RC4

Once this algorithm completes, array *S* still contains all the byte values from 0 to 255 but now in a random-looking order. For example, with the all-zero 128-bit key, the state *S* (from *S*[0] to *S*[255]) becomes:

$$0, 35, 3, 43, 9, 11, 65, 229, (\ldots), 233, 169, 117, 184, 31, 39$$

However, if I flip the first key bit and run the KSA again, I get a totally different, apparently random state:

$$32, 116, 131, 134, 138, 143, 149, (\ldots), 152, 235, 111, 48, 80, 12$$

Given the initial state *S*, RC4 generates a keystream, *KS*, of the same length as the plaintext, *P*, to compute a ciphertext: $C = P \oplus KS$. The bytes of the keystream *KS* are computed from *S* according to the Python code in Listing 5-2, if *P* is *m* bytes long.

```
i = 0
j = 0
for b in range(m):
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]
    KS[b] = S[(S[i] + S[j]) % 256]
```

Listing 5-2: The keystream generation of RC4, where S *is the state initialized in Listing 5-1*

In Listing 5-2, each iteration of the `for` loop modifies up to 2 bytes of RC4's internal state *S*: the *S*[*i*] and *S*[*j*] whose values are swapped. That is, if $i = 0$ and $j = 4$ and if *S*[0] = 56 and *S*[4] = 78, then the swap operation sets *S*[0] to 78 and *S*[4] to 56. If *j* equals *i*, then S[*i*] isn't modified.

This looks too simple to be secure, yet it took 20 years for cryptanalysts to find exploitable flaws. Before the flaws were revealed, we knew RC4's weaknesses only in specific implementations, as in the first Wi-Fi encryption standard, WEP.

RC4 in WEP

WEP, the first-generation Wi-Fi security protocol, is now completely broken due to weaknesses in the protocol's design and in RC4.

In its WEP implementation, RC4 encrypts payload data of 802.11 frames, the datagrams (or packets) that transport data over the wireless network. All payloads delivered in the same session use the same secret key of 40 or 104 bits but have what is a supposedly unique 3-byte nonce encoded in the frame header (the part of the frame that encodes metadata and comes before the actual payload).

The problem is that RC4 doesn't support a nonce, at least not in its official specification, and we can't use a stream cipher without a nonce. The WEP designers addressed this limitation with a workaround: they included a 24-bit nonce in the wireless frame's header and prepended it to the WEP key to be used as RC4's secret key. That is, if the nonce is the bytes *N*[0],

$N[1]$, $N[2]$ and the WEP key is $K[0]$, $K[1]$, $K[2]$, $K[3]$, $K[4]$, the actual RC4 key is $N[0]$, $N[1]$, $N[2]$, $K[0]$, $K[1]$, $K[2]$, $K[3]$, $K[4]$. The net effect is to have 40-bit secret keys yield 64-bit effective keys and to have 104-bit keys yield 128-bit effective keys. The result? The advertised 128-bit WEP protocol actually offers only 104-bit security, at best.

But here are the real problems with WEP's nonce trick:

The nonces are too small at only 24 bits This means that if a nonce is chosen randomly for each new message, you have to wait about $2^{24/2} = 2^{12}$ packets, or a few megabytes' worth of traffic, until you can find two packets encrypted with the same nonce and thus the same keystream. Even if the nonce is a counter running from 0 to $2^{24} - 1$, it takes a few gigabytes' worth of data until a rollover, when the repeated nonce can allow the attacker to decrypt packets. But there's a bigger problem.

Combining the nonce and key in this fashion helps recover the key WEP's three nonsecret nonce bytes let an attacker determine the value of S after three iterations of the key scheduling algorithm. Because of this, cryptanalysts found that the first keystream byte strongly depends on the first secret key byte—the fourth byte ingested by the KSA—and that this bias can be exploited to recover the secret key.

Exploiting those weaknesses requires access to both ciphertexts and the keystream—that is, known or chosen plaintexts. But that's easy enough: known plaintexts occur when the Wi-Fi frames encapsulate data with a known header, and chosen plaintexts occur when the attacker injects known plaintext encrypted with the target key. The upshot is that the attacks work in practice, not just on paper.

Following the appearance of the first attacks on WEP in 2001, researchers found faster attacks that required fewer ciphertexts. Today, you can even find tools such as aircrack-ng that implement the entire attack, from network sniffing to cryptanalysis.

WEP's insecurity is due to both weaknesses in RC4, which takes a single one-use key instead of a key and a nonce (as in any decent stream cipher), and weaknesses in the WEP design itself.

Now let's look at the second biggest failure of RC4.

RC4 in TLS

TLS is the single most important security protocol used on the internet. It's best known for underlying HTTPS connections, but it's also used to protect some virtual private network (VPN) connections, as well as email servers, mobile applications, and many others. And sadly, TLS has long supported RC4.

Unlike WEP, the TLS implementation doesn't make the same blatant mistake of tweaking the RC4 specs in order to use a public nonce. Instead, TLS just feeds RC4 a unique 128-bit session key, which means it's a bit less broken than WEP.

The weakness in TLS is due only to RC4 and its inexcusable flaws: statistical biases, or nonrandomness, which we know is a total deal-breaker for a

stream cipher. For example, the second keystream byte produced by RC4 is zero, with a probability of 1/128, whereas it should ideally be 1/256. (Recall that a byte can take 256 values from 0 to 255; hence, a truly random byte is zero with a chance of 1/256.) Crazier still is the fact that most experts continued to trust RC4 as late as 2013, even though its statistical biases have been known since 2001.

RC4's known statistical biases should have been enough to ditch the cipher altogether, even if we didn't know how to exploit the biases to compromise actual applications. In TLS, RC4's flaws weren't publicly exploited until 2011, but the NSA allegedly managed to exploit RC4's weaknesses to compromise TLS's RC4 connections well before then.

As it turned out, not only was RC4's second keystream byte biased, but all of the first 256 bytes were biased as well. In 2011, researchers found that the probability that one of those bytes comes to zero equals $1/256 + c/256^2$ for some constant, c, taking values between 0.24 and 1.34. It's not just for the byte zero but for other byte values as well. The amazing thing about RC4 is that it fails where even many noncryptographic PRNGs succeed—namely, at producing uniformly distributed pseudorandom bytes (that is, where each of the 256 bytes has a chance of 1/256 of showing up).

RC4's flawed TLS implementation can even be exploited in the weakest attack model, the chosen-ciphertext: you collect ciphertexts and look for the plaintext, not the key. But there's a caveat: you'll need many ciphertexts, encrypting *the same plaintext* several times using different secret keys. We sometimes call this attack model the *broadcast model* because it's akin to broadcasting the same message to multiple recipients.

Say you want to decrypt the plaintext byte P_1 of a plaintext P given many ciphertext bytes obtained by intercepting the different ciphertexts of the same message. You'll thus obtain the first bytes of each of the four ciphertexts $C^1, \ldots, C^4$, for four keystreams $KS^1, \ldots, KS^4$ such that:

$$C_1^1 = P_1 \oplus KS_1^1$$
$$C_1^2 = P_1 \oplus KS_1^2$$
$$C_1^3 = P_1 \oplus KS_1^3$$
$$C_1^4 = P_1 \oplus KS_1^4$$

Because of RC4's bias, keystream bytes KS_1^i (the first byte in each of the four instances) are more likely to be zero than any other byte value. Therefore, C_1^i bytes are more likely to be equal to P_1 than to any other value. To determine P_1 given the C_1^i bytes, simply count the number of occurrences of each byte value and return the most frequent one as P_1. However, because the statistical bias is very small, you need millions of values to get it right with any certainty.

The attack generalizes to recover more than one plaintext byte and to exploit more than one biased value (zero here). The algorithm just becomes a bit more complicated. However, this attack is hard to put into practice because it needs to collect many ciphertexts encrypting the same plaintext but using different keys. For example, the attack can't break all

TLS-protected connections that use RC4 because you need to trick the server into encrypting the same plaintext to many different recipients, or many times to the same recipient with different keys.

Salsa20

Salsa20 is a simple, software-oriented cipher optimized for modern CPUs that has been implemented in numerous protocols and libraries, along with its variant, ChaCha. Its designer, respected cryptographer Daniel J. Bernstein, submitted Salsa20 to the eSTREAM competition in 2005 and won a place in eSTREAM's software portfolio. Salsa20's simplicity and speed have made it popular among developers.

Salsa20 is a counter-based stream cipher—it generates its keystream by repeatedly processing a counter incremented for each block. As Figure 5-10 shows, the *Salsa20 core* algorithm transforms a 512-bit block using a key (K), a nonce (N), and a counter value (Ctr). Salsa20 then adds the result to the original value of the block to produce a *keystream block*. (If the algorithm were to return the core's permutation directly as an output, Salsa20 would be totally insecure because it could be inverted. The final addition of the initial secret state $K \| N \| Ctr$ makes the transform key-to-keystream-block noninvertible.)

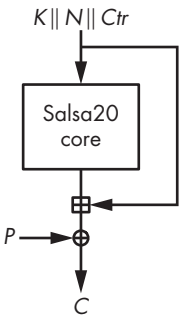


Figure 5-10: Salsa20's encryption scheme for a 512-bit plaintext block

Using the Quarter-Round Function

Salsa20's core permutation uses a function called *quarter-round* (**QR**), which transforms four 32-bit words (a, b, c, and d) as follows:

$$
\begin{aligned}
b &= b \oplus \left[(a + d) <<< 7 \right] \\
c &= c \oplus \left[(b + a) <<< 9 \right] \\
d &= d \oplus \left[(c + b) <<< 13 \right] \\
a &= a \oplus \left[(d + c) <<< 18 \right]
\end{aligned}
$$

We compute these four lines from top to bottom, meaning that the new value of b depends on a and on d, the new value of c depends on a and on the new value of b (and thus d as well), and so on.

The operation <<< is wordwise left-rotation by the specified number of bits, which can be any value between 1 and 31 (for 32-bit words). For example, <<< 8 rotates a word's bits of eight positions toward the left, as the following examples show:

$$\begin{aligned} \text{0x01234567} &<<< 8 = \text{0x23456701} \\ \text{0x01234567} &<<< 16 = \text{0x45670123} \\ \text{0x01234567} &<<< 22 = \text{0x59c048d1} \end{aligned}$$

Transforming Salsa20's 512-Bit State

Salsa20's core permutation transforms a 512-bit internal state viewed as a 4×4 array of 32-bit words. Figure 5-11 shows the initial state, using a key of eight words (256 bits), a nonce of two words (64 bits), a counter of two words (64 bits), and four fixed constant words (128 bits) that are identical for each encryption/decryption and all blocks.

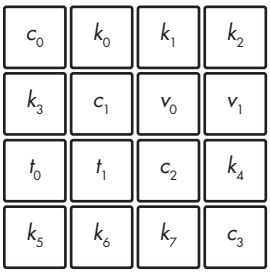


Figure 5-11: The initialization of Salsa20's state

To transform the initial 512-bit state, Salsa20 first applies the **QR** transform to all four columns independently (known as the *column-round*) and then to all four rows independently (the *row-round*), as Figure 5-12 illustrates. The sequence column-round/row-round is a *double-round*. Salsa20 repeats 10 double-rounds, for 20 rounds in total, which is the reason for the *20* in *Salsa20*.

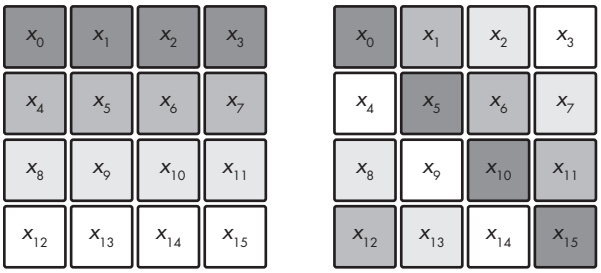


*Figure 5-12: Columns and rows transformed by Salsa20's quarter-round (**QR**) function*

The column-round transforms the four columns like so:

$$\mathbf{QR}(x_0, x_4, x_8, x_{12})$$
$$\mathbf{QR}(x_5, x_9, x_{13}, x_1)$$
$$\mathbf{QR}(x_{10}, x_{14}, x_2, x_6)$$
$$\mathbf{QR}(x_{15}, x_3, x_7, x_{11})$$

The row-round transforms the rows by doing the following:

$$\mathbf{QR}(x_0, x_1, x_2, x_3)$$
$$\mathbf{QR}(x_5, x_6, x_7, x_4)$$
$$\mathbf{QR}(x_{10}, x_{11}, x_8, x_9)$$
$$\mathbf{QR}(x_{15}, x_{12}, x_{13}, x_{14})$$

In a column-round, each **QR** takes x_i arguments ordered from the top to the bottom line, whereas a row-round's **QR** takes as a first argument the words on the diagonal (as in the array on the right in Figure 5-12) rather than words from the first column.

Evaluating Salsa20

Listing 5-3 shows Salsa20's initial states for the first and second blocks when initialized with an all-zero key (`00` bytes) and an all-one nonce (`ff` bytes). These two states differ in only 1 bit, in the counter, which is in bold: specifically, 0 for the first block and 1 for the second.

```
61707865 00000000 00000000 00000000      61707865 00000000 00000000 00000000
00000000 3320646e ffffffff ffffffff      00000000 3320646e ffffffff ffffffff
00000000 00000000 79622d32 00000000      00000001 00000000 79622d32 00000000
00000000 00000000 00000000 6b206574      00000000 00000000 00000000 6b206574
```

Listing 5-3: Salsa20's initial states for the first two blocks with an all-zero key and an all-one nonce

Yet, despite only a 1-bit difference, the respective internal states after 10 double-rounds are totally different from each other, as Listing 5-4 shows.

```
e98680bc f730ba7a 38663ce0 5f376d93      1ba4d492 c14270c3 9fb05306 ff808c64
85683b75 a56ca873 26501592 64144b6d      b49a4100 f5d8fbbd 614234a0 e20663d1
6dcb46fd 58178f93 8cf54cfe cfdc27d7      12e1e116 6a61bc8f 86f01bcb 2efead4a
68bbe09e 17b403a1 38aa1f27 54323fe0      77775a13 d17b99d5 eb773f5b 2c3a5e7d
```

Listing 5-4: The states from Listing 5-3 after 10 Salsa20 double-rounds

But remember, even though word values in the keystream block may look random, it's far from a guarantee of security. RC4's output looks random, but it has blatant biases. Fortunately, Salsa20 is much more secure than RC4 and doesn't have statistical biases. Keep in mind, however, that even when keystreams are statistically indistinguishable from perfectly random bytes, this isn't sufficient to achieve cryptographic security.

Learning Differential Cryptanalysis

To demonstrate why Salsa20 is more secure than RC4, let's look at the basics of *differential cryptanalysis*, the study of the differences between states rather than their actual values. For example, the two initial states in Listing 5-3 differ by 1 bit in the counter or by the word x_8 in the Salsa20 state array. The following array shows the bitwise difference between these two states:

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

The difference between the two states is actually the XOR of these states. The 1 bit in bold corresponds to a 1-bit difference between the two states. In the XOR of the two states, any nonzero bits indicate differences.

To see how fast changes propagate in the initial state as a result of Salsa20's core algorithm, let's look at the difference between two states throughout the rounds iteration. After one round, the difference propagates across the first column to two of the three other words in that column:

```
80040003 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00002000 00000000 00000000 00000000
```

After two rounds, differences further propagate across the rows that already include a difference, which is all but the second row. At this point the differences between the states are rather sparse; not many bits have changed within a word:

```
9ed7eb7f 060002c0 18028b0c 57ca83c0
00000000 00000000 00000000 00000000
00000001 0000e000 801c0006 00000000
00002000 00400000 04000008 0060f300
```

After three rounds, the differences between the states become more dense, though the many zero nibbles indicate that many bit positions are still not affected by the initial difference:

```
3ab3c25d 9f40a5c9 10070e30 07bd03c0
db1ee2ce 43ee9401 21a7022c3 48fd800c
403c1e72 00034003 4dc843be 700b8857
5625b75b 09c00e00 06000348 23f712d4
```

After four rounds, differences look random to a human observer, and they are also almost random statistically as well:

```
d93bed6d a267bf47 760c2f9f 4a41d54b
0e03d792 7340e010 119e6a00 e90186af
7fa9617e b6aca0d7 4f6e9a4a 564b34fd
98be796d 64908d32 4897f7ca a684a2df
```

After only four rounds, a single difference propagates to most of the bits in the 512-bit state. In cryptography, we call this *full diffusion*.

Not only do differences propagate across all states, they also do so according to complex equations that make future differences hard to predict because highly *nonlinear* relations drive the state's evolution, thanks to the mix of XOR, addition, and rotation. If we used only XORs, we'd still have many differences propagating, but the process would be linear and therefore insecure.

Attacking Salsa20/8

Salsa20 makes 20 rounds by default, but we sometimes use it with only 12 rounds, in a version called Salsa20/12, to make it faster. Although Salsa20/12 uses eight fewer rounds than Salsa20, it's in practice as reliable as the 20-round version, according to the state-of-the-art research progress. Even Salsa20/8, with only eight rounds, is known to be only theoretically weaker but as solid in practice as Salsa20.

Breaking Salsa20 should ideally take 2^{256} operations, thanks to its use of a 256-bit key. If one can recover the key by performing fewer than 2^{256} operations, the cipher is in theory broken. That's exactly the case with Salsa20/8.

The attack on Salsa20/8 (published in the 2008 paper "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba," of which I'm a coauthor and for which we won a cryptanalysis prize from Daniel J. Bernstein) exploits a statistical bias in Salsa's core algorithm after four rounds to recover the key of eight-round Salsa20. In reality, this is mostly a theoretical attack: we estimate its complexity at 2^{251} operations of the core function—infeasible, like any computation of, say, 2^{100} operations or more, but less so than breaking the expected 2^{256} complexity.

The attack exploits not only a bias over the first four rounds of Salsa20/8 but also a property of the last four rounds: knowing the nonce, *N*, and the counter, *Ctr* (refer to Figure 5-10), the only value needed to invert the computation from the keystream back to the initial state is the key, *K*. But as Figure 5-13 shows, if you know only some part of *K*, you can partially invert the computation up until the fourth round and observe some bits of that intermediate state—including the biased bit! You'll observe the bias only if you have the correct guess of the partial key; hence, the bias serves as an indicator that you've got the correct key.

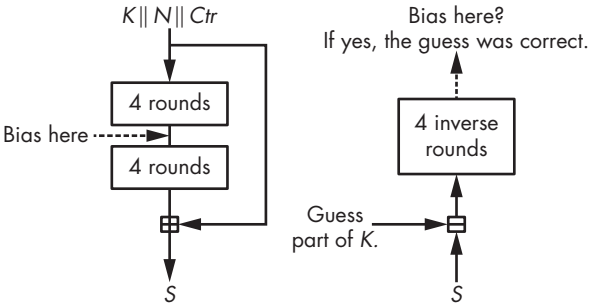


Figure 5-13: The principle of the attack on Salsa20/8

In the actual attack on Salsa20/8, to determine the correct guess we need to guess 220 bits of the key, and we need 2^{31} pairs of keystream blocks, all with the same specific difference in the nonce. Once we single out the correct 220 bits, we brute-force 36 bits. The brute-forcing takes 2^{36} operations, a computation that is dwarfed by the unrealistic $2^{220} \times 2^{31} = 2^{251}$ trials needed to find the 220 bits to complete the first part of the attack.

How Things Can Go Wrong

Alas, many things can go wrong with stream ciphers, from brittle, insecure designs to strong algorithms incorrectly implemented. I'll explore each category of potential problems in the following sections.

Nonce Reuse

The most common failure with stream ciphers occurs when reusing a nonce more than once with the same key. This produces identical keystreams, allowing you to break the encryption—for example, by XORing two ciphertexts together; the keystream then vanishes, and you're left with the XOR of the two plaintexts.

A real example is older versions of Microsoft Word and Excel that used a unique nonce for each document, but modifying a document didn't change the nonce. As a result, one could use the clear and encrypted text of an older version of a document to decrypt later encrypted versions. If Microsoft made this blunder, you can imagine how large the problem might be.

Certain stream ciphers designed in the 2010s tried to mitigate the risk of nonce reuse by building "misuse-resistant" constructions, or ciphers that remain secure even if a nonce is used twice. However, achieving this level of security comes with a performance penalty, as you'll see in Chapter 8 with the SIV mode.

Broken RC4 Implementation

Though it's intrinsically weak, RC4 can become even weaker if you blindly optimize its implementation. For example, let's consider an entry in the 2007 Underhanded C Contest, an informal competition where programmers write benign-looking code that actually includes a malicious function.

Here's how it works. The naive way to implement the line `swap(S[i], S[j])` in RC4's algorithm is to do the following, as this Python code shows:

```
buf = S[i]
S[i] = S[j]
S[j] = buf
```

This way of swapping two variables works, but you need to create a new variable, `buf`. To avoid this, programmers often use the following *XOR-swap* trick to swap the values of the variables `x` and `y`:

```
x = x ⊕ y
y = x ⊕ y
x = x ⊕ y
```

This works because the second line sets `y` to `x ⊕ y ⊕ y = x`, and the third line sets `x` to `x ⊕ y ⊕ x ⊕ y ⊕ y = y`. Using this trick to implement RC4 gives the implementation in Listing 5-5 (adapted from David Wagner and Philippe Biondi's program submitted to the 2007 Underhanded C Contest, and online at *http://www.underhanded-c.org/_page_id_16.html*).

```
#define TOBYTE(x) (x) & 255
#define SWAP(x,y) do { x^=y; y^=x; x^=y; } while (0)

static unsigned char S[256];
static int i=0, j=0;

void init(char *passphrase) {
    int passlen = strlen(passphrase);
    for (i=0; i<256; i++)
        S[i] = i;
    for (i=0; i<256; i++) {
        j = TOBYTE(j + S[TOBYTE(i)] + passphrase[j % passlen]);
        SWAP(S[TOBYTE(i)], S[j]);
    }
    i = 0; j = 0;
}

unsigned char encrypt_one_byte(unsigned char c) {
    int k;
    i = TOBYTE(i+1);
    j = TOBYTE(j + S[i]);
    SWAP(S[i], S[j]);
    k = TOBYTE(S[i] + S[j]);
    return c ^ S[k];
}
```

Listing 5-5: An incorrect C implementation of RC4, due to its use of an XOR swap

Can you spot the problem with the XOR swap?

Things go south when `i = j`. Instead of leaving the state unchanged, the XOR swap sets `S[i]` to `S[i]` ⊕ `S[i] = 0`. In effect, a byte of the state is set to zero each time `i` equals `j` in the key schedule or during encryption, ultimately leading to an all-zero state and thus to an all-zero keystream. For example, after processing 68KB of data, most of the bytes in the 256-byte state are zero, and the output keystream looks like this:

```
00 00 00 00 00 00 00 53 53 00 00 00 00 00 00 00 00 00 00 00 13 13 00 5c 00 a5 00 00 ...
```

The lesson here is to refrain from over-optimizing your crypto implementations. Clarity and confidence always trump performance in cryptography.

Weak Ciphers Baked into Hardware

When a cryptosystem fails to be secure, some systems quickly respond by silently updating the affected software remotely (as with web applications) or by releasing a new version and prompting the users to upgrade (as with mobile applications). Other systems aren't so lucky and need to stick to the compromised cryptosystem for a while before upgrading to a secure version, as is the case with certain satellite phones.

In the early 2000s, US and European telecommunication standardization institutes (TIA and ETSI) jointly developed two standards for satellite phone (satphone) communications. Satphones are like mobile phones, except that their signal goes through satellites rather than terrestrial stations. The advantage is that you can use them pretty much everywhere in the world, as long as you have satellite coverage. Their downsides are the price, quality, latency, and, as it turns out, security.

GMR-1 and GMR-2 are the two satphone standards adopted by most commercial vendors, such as Thuraya and Inmarsat. Both include stream ciphers to encrypt voice communications. GMR-1's cipher is hardware oriented, with a combination of four LFSRs, similar to A5/2, the deliberately insecure cipher in the 2G mobile standard aimed at non-Western countries. GMR-2's cipher is software oriented, with an 8-byte state and the use of S-boxes. Both stream ciphers are insecure and protect users only against amateurs, not against state agencies.

This story reminds us that stream ciphers used to be easier to break than block ciphers and that they're easier to sabotage. Why? Well, if you design a weak stream cipher on purpose, when you find the flaw, you can still blame it on the weakness of stream ciphers and deny any malicious intent.

Further Reading

To learn more about stream ciphers, begin with the archives of the eSTREAM competition at *https://www.ecrypt.eu.org/stream/project.html*, where you'll find hundreds of papers on stream ciphers, including details of more than 30 candidates and many attacks. Some of the most interesting attacks are the

correlation attacks, algebraic attacks, and cube attacks. See in particular the work of Nicolas Courtois and Willi Meier for the first two attack types and that of Itai Dinur and Adi Shamir for cube attacks.

For more about attacks on RC4, look up the 2001 Scott Fluhrer, Itsik Mantin, and Adi Shamir (FMS) attack, and the 2013 research article "On the Security of RC4 in TLS."

Salsa20's legacy deserves your attention, too. The stream cipher ChaCha is similar to Salsa20 but with a slightly different core permutation that was later used in the hash function BLAKE, as you'll see in Chapter 6. These algorithms all leverage Salsa20's software implementation techniques using parallelized instructions, as discussed at *https://cr.yp.to/snuffle.html*.

6

HASH FUNCTIONS

Hash functions—such as SHA-256, SHA3, and BLAKE3—compose the cryptographer's Swiss Army Knife: they are used in digital signatures, public-key encryption, integrity verification, message authentication, password protection, key agreement protocols, and many other cryptographic protocols.

Whether you're encrypting an email, sending a message on your mobile phone, connecting to an HTTPS website, or connecting to a remote machine through a virtual private network (VPN) or Secure Shell (SSH), a hash function is somewhere under the hood.

Hash functions are by far the most versatile and ubiquitous of all crypto algorithms. Their applications include the following: cloud storage systems use them to identify identical files and to detect modified files; the Git revision control system uses them to identify files in a repository; endpoint detection and response (EDR) systems use them to detect modified files; network-based intrusion detection systems (NIDSs) use hashes

to detect known-malicious data going through a network; forensic analysts use hash values to prove that digital artifacts have not been modified; Bitcoin uses a hash function in its proof-of-work systems—and there are many more.

Unlike stream ciphers, which create a long output from a short one, hash functions take a long input and produce a short output, called a *hash value* or *digest* (see Figure 6-1).

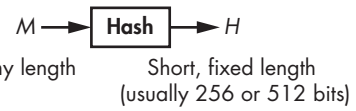


Figure 6-1: A hash function's input and output

This chapter revolves around two main topics. First, security: What does it mean for a hash function to be secure? To that end, I introduce two essential notions—collision resistance and preimage resistance. The second topic revolves around hash function construction. I explain the high-level techniques modern hash functions use and then review the internals of the most common hash functions: SHA-1, SHA-2, SHA-3, and BLAKE2. Finally, you see how secure hash functions can behave insecurely if misused.

NOTE *Don't confuse cryptographic hash functions with* noncryptographic *ones. You use noncryptographic hash functions in data structures such as hash tables or to detect accidental errors, and they provide no security whatsoever. For example, cyclic redundancy checks (CRCs) are noncryptographic hashes you use to detect accidental modifications of a file.*

Secure Hash Functions

The notion of security for hash functions is different from what we've discussed thus far. Whereas ciphers protect data confidentiality in an effort to guarantee that data sent in the clear can't be read, hash functions protect data integrity in an effort to guarantee that data—whether sent in the clear or encrypted—hasn't been modified. If a hash function is secure, two distinct pieces of data should always have different hashes. A file's hash can thus serve as its identifier.

Consider the most common application of a hash function: *digital signatures*, or just *signatures*. When using digital signatures, applications process the hash of the message to be signed rather than the message itself, as Figure 6-2 illustrates.

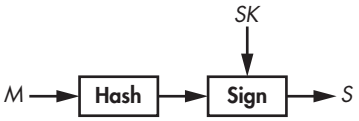


Figure 6-2: A hash function in a digital signature scheme, wherein the hash acts as a proxy for the message

The hash acts as an identifier for the message. If even a single bit is changed in the message, the hash of the message will be totally different. The hash function thus helps ensure that the message hasn't been modified. Signing a message's hash is as secure as signing the message itself, and signing a short hash of, say, 256 bits is much faster than signing a message that may be very large. In fact, most signature algorithms work only on short inputs such as hash values.

Unpredictability Again

The cryptographic strength of hash functions stems from the unpredictability of their outputs. Take the following 256-bit hexadecimal values; you compute these hashes using the NIST standard hash function SHA-256 with the ASCII letters `a`, `b`, and `c` as inputs. Though the values `a`, `b`, and `c` differ by only 1 or 2 bits (`a` is the bit sequence 01100001, `b` is 01100010, and `c` is 01100011), their hash values are completely different:

```
SHA-256("a") = ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
SHA-256("b") = 3e23e8160039594a33894f6564e1b1348bbd7a0088d42c4acb73eeaed59c009d
SHA-256("c") = 2e7d2c03a9507ae265ecf5b5356885a53393a2029d241394997265a1a25aefc6
```

Given only these three hashes, it's impossible to predict the value of the SHA-256 hash of `d` or any of its bits because hash values of a secure hash function are *unpredictable*. A secure hash function should be like a black box that returns a random string each time it receives an input.

The general, theoretical definition of a secure hash function is that it behaves like a truly random function (sometimes called a *random oracle*). Specifically, a secure hash function shouldn't have any property or pattern that a random function wouldn't have. This definition is helpful for theoreticians, but in practice we need more specific notions, namely, preimage resistance and collision resistance.

Preimage Resistance

A *preimage* of a given hash value, H, is any message, M, such that **Hash**(M) = H. Preimage *resistance* describes the security guarantee that given a random hash value, an attacker will never find a preimage of that hash value. Indeed, you can sometimes call hash functions *one-way functions* because you can go from the message to its hash but not the other way around.

Indeed, you can't invert a hash function, even given unlimited computing power. For example, suppose that I hash some message using the SHA-256 hash function and get this 256-bit hash value:

```
f67a58184cef99d6dfc3045f08645e844f2837ee4bfcc6c949c9f76743667adfd
```

Even given infinite time and computing power, you'd never be able to determine *the* message that I picked to produce this particular hash, since there are many messages hashing to the same value. You would therefore find *some* messages that produce this hash value (possibly including the one I picked) but would be unable to determine the message I used. You get *unconditional security*.

For example, there are 2^{256} possible values of a 256-bit hash (a typical length with hash functions used in practice), but there are many more values of, say, 1,024-bit messages (namely, $2^{1,024}$ possible values). Therefore, it follows that, on average, each possible 256-bit hash value will have $2^{1,024} / 2^{256} = 2^{1,024-256} = 2^{768}$ preimages of 1,024 bits each.

In practice, you must be sure that it's practically impossible to find *any* message that maps to a given hash value, not just the message that was used, which is what preimage resistance actually stands for. Specifically, you can speak of first-preimage and second-preimage resistance. *First-preimage resistance* (or just *preimage resistance*) describes cases where it's practically impossible to find a message that hashes to a given value. *Second-preimage resistance*, on the other hand, describes the case that when given a message, M_1, it's practically impossible to find another message, M_2, that hashes to the same value that M_1 does.

The Cost of Preimages

Given a hash function and a hash value, you can search for first preimages by trying different messages until one hits the target hash. Listing 6-1 shows how to do this using an algorithm similar to `solve_preimage()`.

```
solve_preimage(H) {
    repeat {
        M = random_message()
        if Hash(M) == H then return M
    }
}
```

Listing 6-1: The optimal preimage search algorithm for a secure hash function

Here, `random_message()` generates a random message (say, a random 1,024-bit value). If the hash's bit length, n, is large enough, `solve_preimage()` will practically never complete, because it will take on average 2^n attempts before finding a preimage. That's a hopeless situation when working with $n = 256$, as in modern hashes like SHA-256 and BLAKE2.

Why Second-Preimage Resistance Is Weaker

If you can find first preimages, then you can find second preimages as well (for the same hash function). As proof, if the algorithm `solve_preimage()` returns a preimage of a given hash value, use the algorithm in Listing 6-2 to find a second preimage of some message, *M*.

```
solve_second_preimage(M) {
    H = Hash(M)
    return solve_preimage(H)
}
```

Listing 6-2: How to find second preimages if you can find first preimages

You'll find the second preimage by seeing it as a preimage problem and applying the preimage attack. It follows that any second-preimage resistant hash function is also preimage resistant. (Were it not, it wouldn't be second preimage resistant either, per the preceding `solve_second_preimage()` algorithm.) In other words, the best attack you can use to find second preimages is almost identical to the best attack you can use to find first preimages, unless the hash function has some defect that allows for more efficient attacks. Also note that a preimage search attack is fundamentally similar to a key-recovery attack on a block cipher or stream cipher, except that in the case of encryption there is exactly one solution of known size.

Collision Resistance

Whatever hash function you choose, collisions will inevitably exist because of the *pigeonhole principle*, which posits that if you have m holes and n pigeons to put into those holes and if n is greater than m, at least one hole must contain more than one pigeon.

NOTE *You can generalize the pigeonhole principle to other items and containers. For example, any 27-word sequence in the US Constitution includes at least two words that start with the same letter. In the world of hash functions, holes are the hash values, and pigeons are the messages. Because you know that there are many more possible messages than hash values, collisions* must *exist.*

However, despite the inevitable, collisions should be hard to find to consider a hash function *collision resistant*—in other words, attackers shouldn't be able to find two distinct messages that hash to the same value.

The notion of collision resistance relates to that of second-preimage resistance: if you can find second preimages for a hash function, you can also find collisions, as Listing 6-3 shows.

```
solve_collision() {
    M = random_message()
    return (M, solve_second_preimage(M))
}
```

Listing 6-3: The naive collision search algorithm

That is, any collision-resistant hash is also second-preimage resistant. If this were not the case, there would be an efficient solve-second-preimage algorithm that could be used to break collision resistance.

How to Find Collisions

Finding collisions is faster than finding preimages: it takes approximately $2^{n/2}$ operations instead of 2^n, thanks to the *birthday attack*, whose key idea is the following: given N messages and as many hash values, you can produce a total of $N \times (N - 1) / 2$ potential collisions by considering each *pair* of two hash values (a number of the same order of magnitude as N^2). It's called the *birthday* attack because it's usually illustrated using the *birthday paradox*, which is the fact that a group of 23 people will include two people having the same birth date with probability close to 1/2—which is not a paradox, just a surprise for many people.

NOTE $N \times (N - 1) / 2$ *is the count of pairs of two* distinct *messages, where you divide by 2 because you view* (M_1, M_2) *and* (M_2, M_1) *as a same pair, the ordering being unimportant.*

For comparison, in the case of a preimage search, N messages get you only N candidate preimages, whereas the same N messages give approximately N^2 potential collisions. With N^2 instead of N, you can say that there are *quadratically* more chances to find a solution. The complexity of the search is in turn quadratically lower: to find a collision, use the square root of 2^n messages; that is, use $2^{n/2}$ instead of 2^n.

The Naive Birthday Attack

Here's the simplest way to find collisions using the birthday attack:

1. Compute $2^{n/2}$ hashes of $2^{n/2}$ arbitrarily chosen messages and store all the message/hash pairs in a list.
2. Sort the list with respect to the hash value to move any identical hash values next to each other.
3. Search the sorted list to find two consecutive entries with the same hash value.

Unfortunately, this method requires a lot of memory (enough to store $2^{n/2}$ message/hash pairs), and sorting lots of elements slows down the search, requiring about $n2^{n/2}$ operations on average, using a fast sorting algorithm such as quicksort.

Low-Memory Collision Search with the Rho Method

The *Rho method* is an algorithm for finding collisions that, unlike the naive birthday attack, requires only a small amount of memory. It works like this:

1. Given a hash function with n-bit hash values, pick some random hash value (H_1), and define $H_1 = H'_1$.
2. Compute $H_2 = \textbf{Hash}(H_1)$ and $H'_2 = \textbf{Hash}(\textbf{Hash}(H'_1))$. In the first case apply the hash function once, while in the second case apply it twice.
3. Iterate the process and compute $H_{i+1} = \textbf{Hash}(H_i)$, $H'_{i+1} = \textbf{Hash}(\textbf{Hash}(H'_i))$, for increasing values of i, until you reach i such that $H_{i+1} = H'_{i+1}$.

Figure 6-3 helps visualize the attack, where an arrow from, say, H_1 to H_2 means $H_2 = \textbf{Hash}(H_1)$. Observe that the sequence of H_is eventually enters a loop, also called a *cycle*, which resembles the Greek letter rho (ρ) in shape. The cycle starts at H_5 and is characterized by the collision $\textbf{Hash}(H_4) = \textbf{Hash}(H_{10}) = H_5$. The key observation here is that to find a collision, you simply need to find such a cycle. The Rho method allows an attacker to detect the position of the cycle and therefore to find the collision.

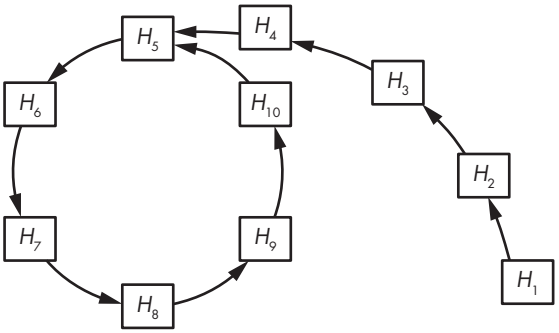


Figure 6-3: The structure of the Rho hash function, where each arrow represents an evaluation of the hash function. The cycle beginning at H_5 *corresponds to a collision,* $\textbf{Hash}(H_4) = \textbf{Hash}(H_{10}) = H_5$.

Advanced collision-finding techniques based on the Rho method work by first detecting the start of the cycle and then finding the collision, without storing numerous values in memory or needing to sort a long list. This takes about $2^{n/2}$ operations to succeed. Indeed, Figure 6-3 has many fewer hash values than would an actual function with digests of 256 bits or more. On average, the cycle and the tail (the part that extends from H_1 to H_5 in Figure 6-3) each include about $2^{n/2}$ hash values, where n is the bit length of the hash values. Therefore, you'll need at least $2^{n/2} + 2^{n/2}$ evaluations of the hash to find a collision.

How to Build Hash Functions

In the 1980s, cryptographers realized that the simplest way to hash a message is to split it into chunks and process each chunk consecutively using a similar algorithm. This strategy, *iterative hashing*, comes in two main forms:

- Iterative hashing using a *compression function* that transforms an input to a *smaller output*, as Figure 6-4 illustrates. This technique is also called the *Merkle–Damgård* construction, named after the cryptographers Ralph Merkle and Ivan Damgård who described it.
- Iterative hashing using a function that transforms an input to an output of the *same size*, such that any two different inputs give two different outputs (that is, a *permutation*). Such functions are called *sponge functions*.

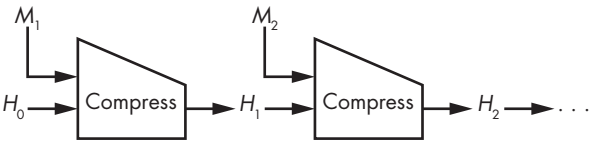


Figure 6-4: The Merkle–Damgård construction using a compression function called Compress

We'll now discuss how these constructions work and how compression functions look in practice.

Compression-Based Hash Functions

All hash functions developed from the 1980s through the 2010s are based on the Merkle–Damgård (M–D) construction: MD4, MD5, SHA-1, and the SHA-2 family, as well as the lesser-known RIPEMD and Whirlpool hash functions. While the M–D construction isn't perfect, it is simple and has proven to be secure enough for many applications.

NOTE *In MD4, MD5, and RIPEMD, the* MD *stands for* message digest *rather than* Merkle–Damgård.

To hash a message, the M–D construction splits the message into blocks of identical size and mixes these blocks with an internal state using a compression function, as Figure 6-4 shows. Here, H_0 is the *initial value* (denoted IV) of the internal state, the values H_1, H_2, . . . are the *chaining values*, and the final value of the internal state is the message's hash value.

The message blocks are often 512 or 1,024 bits, but they can, in principle, be any size. Whatever the block length, it is fixed for a given hash function. For example, SHA-256 works with 512-bit blocks, and SHA-512 works with 1,024-bit blocks.

Padding Blocks

What happens if you want to hash a message that can't be split into a sequence of complete blocks? For example, if blocks are 512 bits, then a 520-bit message consists of one 512-bit block plus 8 bits. In such a case, the M–D construction forms the last block as follows: take the chunk of bits left (8 in our example), append 1 bit, then append 0 bits, and finally append the length of the original message, encoded on a fixed number of bits. This padding trick guarantees that any two distinct messages give a distinct sequence of blocks and thus a distinct hash value.

For example, if you hash the 8-bit string 10101010 using SHA-256, which is a hash function with 512-bit message blocks, the first and only block appears, in bits, as follows:

$$10101010\ \textit{1000000000000000}\ (\ldots)\ 000000000000\underline{\textit{1000}}$$

Here, the message bits are the first 8 bits (10101010), and the padding bits are all the subsequent bits (shown in italic). The *1000* at the end of the block (underlined) is the message's length, or 8 encoded in binary (on 32 bits at most). The padding thus produces a 512-bit message composed of a single 512-bit block, ready to be processed by SHA-256's compression function.

Security Guarantees

The Merkle–Damgård construction turns a secure compression function that takes small, fixed-length inputs into a secure hash function that takes inputs of arbitrary lengths. If a compression function is preimage and collision resistant, then a hash function built on it using the M–D construction is also preimage and collision resistant. This is true because we can turn any successful preimage attack for the M–D hash into a successful preimage attack for the compression function, as Merkle and Damgård demonstrated in their 1989 papers (see this chapter's "Further Reading" section). The same is true for collisions: an attacker can't break the hash's collision resistance without breaking the underlying compression function's collision resistance; hence, the security of the latter guarantees the security of the hash.

Note that the converse argument doesn't hold, because a collision for the compression function doesn't necessarily give a collision for the hash. An arbitrary collision between **Compress**(X, M_1) and **Compress**(Y, M_2) for chaining values X and Y, both distinct from H_0, won't get you a collision for the hash because you can't "plug" the collision into the iterative chain of hashes—except if one of the chaining values happens to be X and the other Y, but that's unlikely to happen.

Finding Multicollisions

A *multicollision* occurs when a set of three or more messages hash to the same value. For example, the triplet (X, Y, Z), such that **Hash**(X) = **Hash**(Y) = **Hash**(Z), is a *3-collision*. Ideally, multicollisions should be much harder to

find than collisions, but there's a simple trick for finding them at almost the same cost as that of a single collision. Here's how it works:

1. Find a collision **Compress**(H_0, $M_{1.1}$) = **Compress**(H_0, $M_{1.2}$) = H_1. This is just a 2-collision, or two messages hashing to the same value.
2. Find a second collision with H_1 as a starting chaining value: **Compress** (H_1, $M_{2.1}$) = **Compress**(H_1, $M_{2.2}$) = H_2. Now you have a 4-collision, with four messages hashing to the same value H_2: $M_{1.1} \parallel M_{2.1}$, $M_{1.1} \parallel M_{2.2}$, $M_{1.2} \parallel M_{2.1}$, and $M_{1.2} \parallel M_{2.2}$.
3. Repeat and find N times a collision, and you'll have 2^N messages, each of N blocks, hashing to the same value—that is, a 2^N-collision, at the cost of "only" about $N2^N$ hash computations.

In practice, this trick isn't all that practical because it requires finding a basic 2-collision in the first place.

Building Compression Functions

All compression functions used in real hash functions such as SHA-256 and BLAKE2 are based on block ciphers because that's the simplest way to build a compression function. Figure 6-5 shows the most common of the block cipher–based compression functions, the *Davies–Meyer construction*.

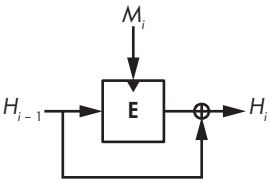


Figure 6-5: The Davies–Meyer construction. The dark triangle shows where the block cipher's key is input.

Given a message block M_i and the previous chaining value H_{i-1}, the Davies–Meyer compression function uses a block cipher, **E**, to compute the new chaining value as:

$$H_i = \mathbf{E}(M_i, H_{i-1}) \oplus H_{i-1}$$

The message block M_i acts as the block cipher key, and the chaining value H_{i-1} acts as its plaintext block. As long as the block cipher is secure, the resulting compression function is secure as well as collision and preimage resistant. Without the XOR of the preceding chaining value ($\oplus H_{i-1}$), Davies–Meyer would be insecure because you could invert it, going from the new chaining value to the previous one using the block cipher's decryption function.

NOTE *The Davies–Meyer construction has a surprising property: you can find* fixed points*, or chaining values, that are unchanged after applying the compression function with a given message block. It suffices to take* $H_{i-1} = \mathbf{D}(M_i, 0)$ *as a chaining value, where* **D** *is the decryption function corresponding to* ***E****. The new chaining value* H_i *is therefore equal to the original* H_{i-1}*:*

$$H_i = \mathbf{E}\big(M_i, H_{i-1}\big) \oplus H_{i-1} = \mathbf{E}\Big(M_i, \mathbf{D}\big(M_i, 0\big)\Big) \oplus \mathbf{D}\big(M_i, 0\big)$$
$$= 0 \oplus \mathbf{D}\big(M_i, 0\big) = \mathbf{D}\big(M_i, 0\big) = H_{i-1}$$

You get $H_i = H_{i-1}$ *because plugging the decryption of zero into the encryption function yields zero—the term* $\boldsymbol{E}(M_i, \boldsymbol{D}(M_i, 0)) = 0$*—leaving only the* $\oplus H_{i-1}$ *part of the equation in the expression of the compression function's output. You can then find fixed points for the compression functions of the SHA-2 functions, for example, which are based on the Davies–Meyer construction. Fortunately, fixed points aren't a security risk.*

There are many block cipher–based compression functions other than Davies–Meyer, such as those in Figure 6-6.

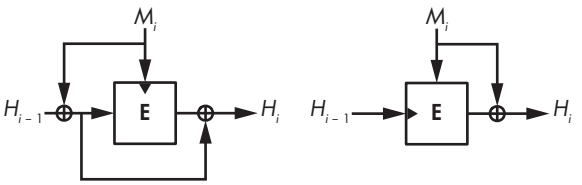


Figure 6-6: Other secure block cipher–based compression function constructions

These are less popular because they're more complex or require the message block to be the same length as the chaining value.

Permutation-Based Hash Functions

After decades of research, cryptographers know everything there is to know about block cipher–based hashing techniques. Still, shouldn't there be a simpler way to hash? Why bother with a block cipher, an algorithm that takes a secret key, when hash functions don't take a secret key? Why not build hash functions with a fixed-key block cipher, a single permutation algorithm?

Those simpler hash functions are sponge functions, and they use a single permutation instead of a compression function and a block cipher (see Figure 6-7). Instead of using a block cipher to mix message bits with the internal state, sponge functions just do an XOR operation. Sponge functions are not only simpler than Merkle–Damgård functions, they're also more versatile. You'll find them used as hash functions and also as deterministic random bit generators, stream ciphers, pseudorandom

functions (see Chapter 7), and authenticated ciphers (see Chapter 8). The most famous sponge function is Keccak, also known as SHA-3.

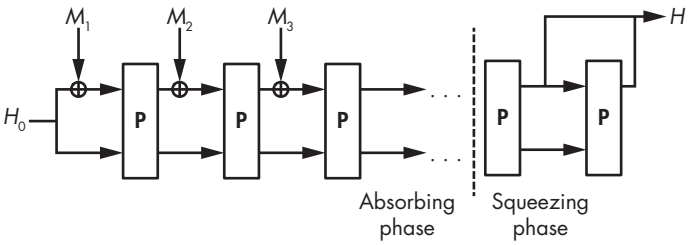


Figure 6-7: The sponge construction

A sponge function works as follows:

1. It XORs the first message block, M_1, to H_0, a predefined initial value of the internal state (for example, the all-zero string). Message blocks are all the same size and smaller than the internal state.
2. A permutation, **P**, transforms the internal state to another value of the same size.
3. It XORs block M_2 and applies **P** and then repeats this for the message blocks M_3, M_4, and so on. This is the *absorbing phase*.
4. After injecting all the message blocks, it applies **P** again and extracts a block of bits from the state to form the hash. If you need a longer hash, apply **P** again and extract a block. This is the *squeezing phase*.

The security of a sponge function depends on the length of its internal state and the length of the blocks. If message blocks are r-bit long and the internal state is w-bit long, then there are $c = w - r$ bits of the internal state that message blocks can't modify. The value of c is a sponge's *capacity*, and the security level guaranteed by the sponge function is $c/2$. For example, to reach 256-bit security with 64-bit message blocks, the internal state should be at least $w = 2 \times 256 + 64 = 576$ bits. The security level also depends on the length, n, of the hash value. The complexity of a collision attack is therefore the smallest value between $2^{n/2}$ and $2^{c/2}$, while the complexity of a second preimage attack is the smallest value between 2^n and $2^{c/2}$.

To be secure, the permutation **P** should behave like a random permutation, without statistical bias and without a mathematical structure that would allow an attacker to predict outputs. As with compression function–based hashes, sponge functions also pad messages, but the padding is simpler because it doesn't need to include the message's length. The last message bit is simply followed by a 1 bit and as many zeros as necessary.

The SHA Family of Hash Functions

The *Secure Hash Algorithm (SHA)* hash functions are standards defined by NIST for use by nonmilitary federal government agencies in the United

States. They are considered worldwide standards, and only certain non-US governments opt for their own hash algorithms (such as China's SM3, Russia's Streebog, and Ukraine's Kupyna) for reasons of sovereignty rather than a lack of trust in SHA's security. The US SHAs have been more extensively reviewed by cryptanalysts than the non-US ones.

NOTE *Message Digest 5 (MD5) was the most popular hash function from 1992 until it was broken around 2005, and many applications switched to one of the SHA hash functions. MD5 processes 512-bit block messages and updates a 128-bit internal state to produce a 128-bit hash, thus providing at best 128-bit preimage security and 64-bit collision security. In 1996, cryptanalysts warned of a collision for MD5's compression function, but their warning went unheeded until 2005 when a team of Chinese cryptanalysts discovered how to compute collisions for the full MD5 hash. As I write this, it takes only seconds to find a collision for MD5, yet some systems still use or support MD5, often for reasons of backward compatibility.*

SHA-1

The SHA-1 standard arose from a failure in the NSA's original SHA-0 hash function. In 1993, NIST standardized the SHA-0 hash algorithm, but in 1995 the NSA released SHA-1 to fix an unidentified security issue in SHA-0. The reason for the tweak became clear when in 1998 two researchers discovered how to find collisions for SHA-0 in about 2^{60} operations instead of the 2^{80} expected for 160-bit hash functions such as SHA-0 and SHA-1. Later attacks reduced the complexity to around 2^{33} operations, leading to actual collisions in less than an hour for SHA-0.

SHA-1 Internals

SHA-1 combines a Merkle–Damgård hash function with a Davies–Meyer compression function based on a specially crafted block cipher. That is, SHA-1 works by iterating the following operation over 512-bit message blocks (M):

$$H = \mathbf{E}\big(M, H\big) + H$$

Here, the use of a plus sign (+) rather than ⊕ (XOR) is intentional. $\mathbf{E}(M, H)$ and H are viewed as arrays of 32-bit integers, and two words at the same position are added together: the first 32-bit word of $\mathbf{E}(M, H)$ with the first 32-bit word of H, and so on. The initial value of H is constant for any message, then H is modified as per the previous equation, and the final value of H after processing all blocks is returned as the hash of the message.

Once the block cipher is run using the message block as a key and the current 160-bit chaining value as a plaintext block, the 160-bit result is seen as an array of five 32-bit words, each of which is added to its 32-bit counterpart in the initial H value.

Listing 6-4 shows SHA-1's compression function, `SHA1-compress()`:

```
SHA1-compress(H, M) {
    (a0, b0, c0, d0, e0) = H   // Parsing H as five 32-bit big-endian words
    (a, b, c, d, e) = SHA1-blockcipher(a0, b0, c0, d0, e0, M)
    return (a + a0, b + b0, c + c0, d + d0, e + e0)
}
```

Listing 6-4: SHA-1's compression function

SHA-1's block cipher `SHA1-blockcipher()`, shown in bold, takes a 512-bit message block, *M*, as a key and transforms the five 32-bit words (`a`, `b`, `c`, `d`, and `e`) by iterating 80 steps of a short sequence of operations to replace the word `a` with a combination of all five words. It then shifts the other words in the array, as in a shift register. Listing 6-5 describes these operations in pseudocode, where `K[i]` is a round-dependent constant.

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

Listing 6-5: SHA-1's block cipher

The `expand()` function in Listing 6-5 creates an array of 80 32-bit words, *W*, from the 16-word message block by setting *W*'s first 16 words to *M* and the subsequent ones to an XOR combination of previous words, rotated 1 bit to the left. Listing 6-6 shows the corresponding pseudocode.

```
expand(M) {
    // The 512-bit M is seen as an array of sixteen 32-bit words.
    W = empty array of eighty 32-bit words
    for i = 0 to 79 {
        if i < 16 then W[i] = M[i]
        else
            W[i] = (W[i - 3] ⊕ W[i - 8] ⊕ W[i - 14] ⊕ W[i - 16]) <<< 1
    }
    return W
}
```

Listing 6-6: SHA-1's expand() function

The `<<< 1` operation in Listing 6-6 is the only difference between the SHA-1 and the SHA-0 functions.

Finally, Listing 6-7 shows the `f()` function in `SHA1-blockcipher()`, a sequence of basic bitwise logical operations (a Boolean function) that depends on the round number.

```
f(i, b, c, d) {
    if i < 20 then return ((b & c) ⊕ (~b & d))
    if i < 40 then return (b ⊕ c ⊕ d)
    if i < 60 then return ((b & c) ⊕ (b & d) ⊕ (c & d))
    if i < 80 then return (b ⊕ c ⊕ d)
}
```

Listing 6-7: SHA-1's `f()` function

The second and fourth Boolean functions in Listing 6-7 simply XOR the three input words together, which is a linear operation. In contrast, the first and third functions use the nonlinear `&` operator (logical AND) to protect against differential cryptanalysis, which exploits the predictable propagation of bitwise difference. Without the `&` operator (in other words, if `f()` were always `b` ⊕ `c` ⊕ `d`, for example), SHA-1 would be easy to break by tracing patterns within its internal state.

Attacks on SHA-1

Though more robust than SHA-0, SHA-1 is also insecure, which is why the Chrome browser marks websites using SHA-1 in their HTTPS connection as insecure since 2014. Although its 160-bit hash should grant it 80-bit collision resistance, in 2005 researchers found weaknesses in SHA-1 and estimated that finding a collision would take approximately 2^{63} calculations (against 2^{80} if the algorithm were flawless). A real SHA-1 collision came only 12 years later when after years of research, cryptanalysts presented two colliding PDF documents through a joint work with Google researchers (see *https://shattered.io*).

You should not use SHA-1. Most web browsers now mark SHA-1 as insecure, and SHA-1 is no longer recommended by NIST. Use SHA-2, SHA-3, BLAKE2, or BLAKE3 instead.

SHA-2

SHA-2, the successor to SHA-1, was designed by the NSA and standardized by NIST in 2002. SHA-2 is a family of four hash functions: SHA-224, SHA-256, SHA-384, and SHA-512 (of which SHA-256 and SHA-512 are the two main algorithms). The three-digit numbers represent the bit lengths of each hash.

The initial motivation behind the development of SHA-2 was to generate longer hashes and thus deliver higher security levels than SHA-1. However, SHA-1 and SHA-2 algorithms are similar in their construction. All SHA-2 instances also use the Merkle–Damgård construction and have a compression function that closely resembles that of SHA-1 but with stronger nonlinearity and difference propagation properties.

SHA-256

SHA-256 is the most common version of SHA-2. Whereas SHA-1 has 160-bit chaining values, SHA-256 has 256-bit chaining values as eight 32-bit words. Both SHA-1 and SHA-256 have 512-bit message blocks, but whereas SHA-1 makes 80 rounds, SHA-256 makes 64 rounds, expanding the 16-word

message block to a 64-word message block using the `expand256()` function, as Listing 6-8 shows.

```
expand256(M) {
    // The 512-bit M is seen as an array of sixteen 32-bit words.
    W = empty array of sixty-four 32-bit words
    for i = 0 to 63 {
        if i < 16 then W[i] = M[i]
        else {
            // The ">>" shifts instead of a ">>>" rotates and is not a typo.
            s0 = (W[i - 15] >>> 7) ⊕ (W[i - 15] >>> 18) ⊕ (W[i - 15] >> 3)
            s1 = (W[i - 2] >>> 17) ⊕ (W[i - 2] >>> 19) ⊕ (W[i - 2] >> 10)
            W[i] = W[i - 16] + s0 + W[i - 7] + s1
        }
    }
    return W
}
```

Listing 6-8: SHA-256's `expand256()` function

Note how SHA-2's `expand256()` message expansion is more complex than SHA-1's `expand()` in Listing 6-6, which in contrast simply performs XORs and a 1-bit rotation. The main loop of SHA-256's compression function is also more complex than that of SHA-1, performing 26 arithmetic operations per iteration compared to 11 for SHA-1. Again, these operations are XORs, logical ANDs, and word rotations. This greater complexity makes SHA-256 more resistant to differential cryptanalysis.

Other SHA-2 Algorithms

The SHA-2 family includes SHA-224, which is algorithmically identical to SHA-256 except that its initial value is a different set of eight 32-bit words, and its hash value length is 224 bits, instead of 256 bits, and is taken as the first 224 bits of the final chaining value.

The SHA-2 family also includes the algorithms SHA-512 and SHA-384. SHA-512 is similar to SHA-256 except that it works with 64-bit words instead of 32-bit words. As a result, it uses 512-bit chaining values (eight 64-bit words) and ingests 1,024-bit message blocks (sixteen 64-bit words), and it makes 80 rounds instead of 64. The compression function is otherwise almost the same as that of SHA-256, though with different rotation distances to cope with the wider word size. (For example, SHA-512 includes the operation `a >>> 34`, which wouldn't make sense with SHA-256's 32-bit words.) SHA-384 is to SHA-512 what SHA-224 is to SHA-256—namely, the same algorithm but with a different initial value and a final hash truncated to 384 bits.

Security-wise, all four SHA-2 versions have lived up to their promises so far: SHA-256 guarantees 256-bit preimage resistance, SHA-512 guarantees about 256-bit collision resistance, and so on. Still, there is no genuine proof that SHA-2 functions are secure; we're talking about probable security.

That said, after practical attacks on MD5 and on SHA-1, researchers and NIST grew concerned about SHA-2's long-term security because of

its similarity to SHA-1, and many believed that attacks on SHA-2 were just a matter of time. As I write this, we have yet to see a successful attack on SHA-2. Regardless, NIST developed a backup plan: SHA-3.

The SHA-3 Competition

Announced in 2007, the NIST Hash Function Competition (the official name of the SHA-3 competition) began with a call for submissions and some basic requirements: hash submissions were to be at least as secure and as fast as SHA-2, and they should be able to do at least as much as SHA-2. SHA-3 candidates also shouldn't look too much like SHA-1 and SHA-2 to be immune to attacks that would break SHA-1 and potentially SHA-2. By 2008, NIST had received 64 submissions from around the world, including from universities and large corporations (BT, IBM, Microsoft, Qualcomm, and Sony, to name a few). Of these 64 submissions, 51 matched the requirements and entered the first round of the competition.

During the first weeks of the competition, cryptanalysts mercilessly attacked the submissions. In July 2009, NIST announced 14 second-round candidates. After spending 15 months analyzing and evaluating the performance of these candidates, NIST chose five finalists:

BLAKE An enhanced Merkle–Damgård hash whose compression function is based on a block cipher, which is in turn based on the core function of the stream cipher ChaCha, a chain of additions, XORs, and word rotations. BLAKE was designed by a team of academic researchers based in Switzerland and the UK, including myself while I was a PhD student.

Grøstl An enhanced Merkle–Damgård hash whose compression function uses two permutations (or fixed-key block ciphers) based on the AES block cipher. Grøstl was designed by a team of seven academic researchers from Denmark and Austria.

JH A tweaked sponge function construction wherein message blocks are injected before and after the permutation rather than just before. The permutation also performs operations similar to a substitution–permutation block cipher (see Chapter 4). JH was designed by a cryptographer from a university in Singapore.

Keccak A sponge function whose permutation performs only bitwise operations. Keccak was designed by a team of four cryptographers working for a semiconductor company based in Belgium and Italy and included one of the two designers of AES.

Skein A hash function based on a different mode of operation than Merkle–Damgård and whose compression function is based on a novel block cipher that uses only integer addition, XORs, and word rotation. Skein was designed by a team of eight cryptographers from academia and industry, all but one of whom is based in the United States, including the renowned Bruce Schneier.

After extensive analysis of the five finalists, NIST announced a winner: Keccak. NIST's report rewarded Keccak for its "elegant design, large security margin, good general performance, excellent efficiency in hardware, and its flexibility." Let's see how Keccak works.

Keccak (SHA-3)

One of the reasons that NIST chose Keccak is that it's completely different from SHA-1 and SHA-2. For one thing, it's a sponge function. Keccak's core algorithm is a permutation of a 1,600-bit state that ingests blocks of 1,152, 1,088, 832, or 576 bits, producing hash values of 224, 256, 384, or 512 bits, respectively—the same four lengths produced by SHA-2 hash functions. But unlike SHA-2, SHA-3 uses a single core algorithm rather than two algorithms for all four hash lengths.

Another reason is that Keccak is more than just a hash. The SHA-3 standard document FIPS 202 defines four hashes—SHA3-224, SHA3-256, SHA3-384, and SHA3-512—and two algorithms called SHAKE128 and SHAKE256. (The name *SHAKE* stands for *Secure Hash Algorithm with Keccak*.) These two algorithms are *extendable-output functions (XOFs)*, or hash functions that can produce hashes of variable length, even very long ones. The numbers 128 and 256 represent the security level of each algorithm.

The FIPS 202 standard itself is lengthy and hard to parse, but you'll find open source implementations that are reasonably fast and make the algorithm easy to understand. For example, the tiny_sha3 (*https://github.com/mjosaarinen/tiny_sha3*) by Markku-Juhani O. Saarinen explains Keccak's core algorithm in 19 lines of C code, as partially reproduced in Listing 6-9.

```
static void sha3_keccakf(uint64_t st[25], int rounds)
{
    (⊕)
    for (r = 0; r < rounds; r++) {

      ❶ // Theta
        for (i = 0; i < 5; i++)
            bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];

        for (i = 0; i < 5; i++) {
            t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
            for (j = 0; j < 25; j += 5)
                st[j + i] ^= t;
        }

      ❷ // Rho Pi
        t = st[1];
        for (i = 0; i < 24; i++) {
            j = keccakf_piln[i];
            bc[0] = st[j];
            st[j] = ROTL64(t, keccakf_rotc[i]);
            t = bc[0];
        }
```

```
    ❸ // Chi
      for (j = 0; j < 25; j += 5) {
          for (i = 0; i < 5; i++)
              bc[i] = st[j + i];
          for (i = 0; i < 5; i++)
              st[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
      }

    ❹ // Iota
      st[0] ^= keccakf_rndc[r];
  }
  (⊕)
}
```

Listing 6-9: The tiny_sha3 implementation

The tiny_sha3 program implements the permutation, **P**, of Keccak, an invertible transformation of a 1,600-bit state viewed as an array of twenty-five 64-bit words. The code iterates a series of rounds, where each round consists of four main steps:

1. `Theta` ❶ includes XORs between 64-bit words or a 1-bit rotated value of the words (the `ROTL64(w, 1)` operation left-rotates a word `w` of 1 bit).
2. `Rho Pi` ❷ includes rotations of 64-bit words by constants hardcoded in the `keccakf_rotc[]` array.
3. `Chi` ❸ includes more XORs but also logical ANDs (the `&` operator) between 64-bit words. These ANDs are the only nonlinear operations in Keccak, and they bring with them cryptographic strength.
4. `Iota` ❹ includes an XOR with a 64-bit constant, hardcoded in `keccakf_rndc[]`.

These operations provide SHA-3 with a strong permutation algorithm free of any bias or exploitable structure. SHA-3 is the product of more than a decade of research, and hundreds of skilled cryptanalysts have failed to break it. It's unlikely to be broken anytime soon.

The BLAKE2 and BLAKE3 Hash Functions

Security may matter most, but speed comes second. I've seen many cases where a developer wouldn't switch from MD5 to SHA-1 simply because MD5 is faster or from SHA-1 to SHA-2 because SHA-2 is noticeably slower than SHA-1. Unfortunately, SHA-3 isn't faster than SHA-2, and because SHA-2 is still secure, there are few incentives to upgrade to SHA-3. So how do we hash faster than SHA-1 and SHA-2 and be even more secure? The answer lies in the hash function BLAKE2, released after the SHA-3 competition.

NOTE *Full disclosure: I'm a designer of BLAKE2, together with Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein.*

BLAKE2 was designed with the following ideas in mind:

- It should be at least as secure as SHA-3, if not stronger.
- It should be faster than all previous hash standards, including MD5.
- It should be suited for use in modern applications and able to hash large amounts of data either as a few large messages or as many small ones, with or without a secret key.
- It should be suited for use on modern CPUs supporting parallel computing on multicore systems as well as instruction-level parallelism within a single core.

The outcome of the engineering process is a pair of main hash functions:

- BLAKE2b (or just BLAKE2), optimized for 64-bit platforms, produces digests ranging from 1 to 64 bytes.
- BLAKE2s, optimized for 8- to 32-bit platforms, produces digests ranging from 1 to 32 bytes.

Each function has a parallel variant that can leverage multiple CPU cores. The parallel counterpart of BLAKE2b, BLAKE2bp, runs on four cores, whereas BLAKE2sp runs on eight cores. The former is the fastest on modern server and laptop CPUs and can hash at close to 2Gbps on a laptop CPU. BLAKE2's speed and features have made it the most popular non-NIST-standard hash. BLAKE2 is used in countless software applications and has been integrated into major cryptography libraries such as OpenSSL and Sodium. BLAKE2 is also an Internet Engineering Task Force (IETF) standard, described in RFC 7693.

NOTE *You can find BLAKE2's specifications and reference code at* https://blake2.net, *and you can download optimized code and libraries from* https://github.com/BLAKE2. *The reference code also provides BLAKE2X, an extension of BLAKE2 that can produce hash values of arbitrary length.*

As Figure 6-8 illustrates, BLAKE2's compression function is a variant of the Davies–Meyer construction that takes parameters as additional input—namely, a *counter* (which ensures that each compression function behaves like a different function) and a *flag* (which indicates whether the compression function is processing the last message block, for increased security).

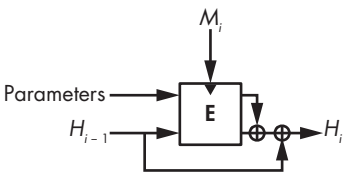


Figure 6-8: BLAKE2's compression function. The two halves of the state are XORed together after the block cipher.

The block cipher in BLAKE2's compression function is based on the stream cipher ChaCha, itself a variant of the Salsa20 stream cipher discussed in Chapter 5. Within this block cipher, BLAKE2b's core operation is composed of the following chain of operations, which transforms a state of four 64-bit words using two message words, M_i and M_j:

$$
\begin{aligned}
a &= a + b + M_i \\
d &= \left((d \oplus a) >>> 32\right) \\
c &= c + d \\
b &= \left((b \oplus c) >>> 24\right) \\
a &= a + b + M_j \\
d &= \left((d \oplus a) >>> 16\right) \\
c &= c + d \\
b &= \left((b \oplus c) >>> 63\right)
\end{aligned}
$$

BLAKE2s's core operation is similar but works with 32-bit instead of 64-bit words (and thus uses different rotation values).

Last but not least, BLAKE3 is a more parallelizable, simpler, more versatile, and faster version of BLAKE2 that was presented in 2020 at the Real World Crypto conference. Designed by Jack O'Connor, Samuel Neves, Zooko Wilcox-O'Hearn, and myself, BLAKE3 has quickly become one of the most popular hash functions, thanks to its undeniable advantages. For more details, see *https://github.com/BLAKE3-team/BLAKE3*.

How Things Can Go Wrong

Despite their apparent simplicity, hash functions can cause major security troubles when used at the wrong place or in the wrong way—for example, when using weak checksum algorithms like CRCs instead of a crypto hash to check file integrity in applications transmitting data over a network. However, this weakness pales in comparison to others, which can cause total compromise in seemingly secure hash functions. You'll see two examples of failures: the first one applies to SHA-1 and SHA-2 but not to BLAKE2 or SHA-3, whereas the second one applies to all four of these functions.

The Length-Extension Attack

Figure 6-9 shows the *length-extension attack*, which is the main threat to the Merkle–Damgård construction.

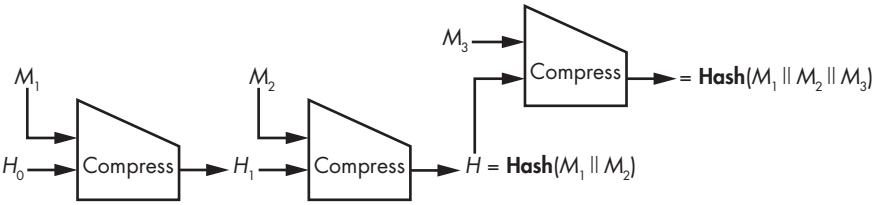


Figure 6-9: The length-extension attack

Basically, if you know **Hash**(M) for some *unknown* message, M, composed of blocks M_1 and M_2 (after padding), you can determine **Hash**(M_1 || M_2 || M_3) for any block, M_3. Because the hash of M_1 || M_2 is the chaining value that follows immediately after M_2, you can add another block, M_3, to the hashed message, even though you don't know the data that was hashed. What's more, this trick generalizes to any number of blocks in the unknown message (M_1 || M_2 here) or in the suffix (M_3).

The length-extension attack won't affect most applications of hash functions, but it can compromise security if using the hash a bit too creatively. Unfortunately, SHA-2 hash functions are vulnerable to the length-extension attack, even though the NSA designed the functions and NIST standardized them while both were well aware of the flaw. This flaw could have been avoided simply by making the last compression function call different from all others (for example, by taking a 1 bit as an extra parameter while the previous calls take a 0 bit). That is what BLAKE2 does.

Fooling Proof-of-Storage Protocols

Cloud computing applications use hash functions within *proof-of-storage* protocols—that is, protocols where a server (the cloud provider) proves to a client (a user of a cloud storage service) that the server does in fact store the files that it's supposed to store on behalf of the client.

In 2007, the paper "SafeStore: A Durable and Practical Storage System" (*https://www.cs.utexas.edu/~lorenzo/papers/p129-kotla.pdf*) by Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin proposed a proof-of-storage protocol to verify the storage of some file, M, as follows:

1. The client picks a random value, C, as a *challenge*.
2. The server computes **Hash**(M || C) as a *response* and sends the result to the client.
3. The client also computes **Hash**(M || C) and checks that it matches the value received from the server.

The premise of the paper is that the server shouldn't be able to fool the client because if the server doesn't know M, it can't guess **Hash**(M || C). But there's a catch: in reality, **Hash** is an iterated hash that processes its input block by block, computing intermediate chaining values between each block. For example, if **Hash** is SHA-256 and M is 512 bits long (the size of a block in SHA-256), the server can cheat. How? The first time the

server receives M, it computes H_1 = **Compress**(H_0, M_1), the chaining value obtained from SHA-256's initial value, H_0, and from the 512-bit M. It then records H_1 in memory and discards M, at which point it no longer stores M.

When the client sends a random value, C, the server computes **Compress**(H_1, C), after adding the padding to C to fill a complete block, and returns the result as **Hash**($M \,||\, C$). The client then believes that because the server returned the correct value of **Hash**($M \,||\, C$), it holds the complete message—except that it may not, as you've seen.

This trick works for SHA-1, SHA-2, as well as SHA-3 and BLAKE2. The solution is simple: ask for **Hash**($C \,||\, M$) instead of **Hash**($M \,||\, C$).

Further Reading

To learn more about hash functions, read the classics from the 1980s and '90s: research articles like Ralph Merkle's "One Way Hash Functions and DES" and Ivan Damgård's "A Design Principle for Hash Functions." Also read the first thorough study of block cipher–based hashing, "Hash Functions Based on Block Ciphers: A Synthetic Approach" by Bart Preneel, René Govaerts, and Joos Vandewalle.

For more on collision search, read the 1997 paper "Parallel Collision Search with Cryptanalytic Applications" by Paul van Oorschot and Michael Wiener. To learn more about the theoretical security notions that underpin preimage resistance and collision resistance, as well as length-extension attacks, search for *indifferentiability*.

For more recent research on hash functions, see the archives of the SHA-3 competition, which include all the different algorithms and how they were broken. You'll find many references on the SHA-3 Zoo at *https://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.html*, and on NIST's page, *https://csrc.nist.gov/projects/hash-functions/sha-3-project*.

For more on the SHA-3 winner Keccak and sponge functions, see *https://keccak.team/sponge_duplex.html*, the official page of the Keccak designers.

Finally, you may look up these two real exploitations of weak hash functions:

- The nation-state malware Flame exploited an MD5 collision to make a counterfeit certificate and appear to be a legitimate piece of software.
- The Xbox game console used a weak block cipher (called TEA) to build a hash function, which was exploited to hack the console and run arbitrary code on it.

7

KEYED HASHING

The hash functions in Chapter 6 take a message and return its hash value—typically a short string of 256 or 512 bits. Anyone can compute the hash value of a message and verify that a particular message hashes to a particular value. When you want only specific people to compute hashes, however, you'll hash with secret keys using *keyed* hash functions.

Keyed hashing forms the basis of two types of cryptographic algorithms: *message authentication codes (MACs)*, which authenticate a message and protect its integrity, and *pseudorandom functions (PRFs)*, which produce random-looking hash-sized values. We'll look at the similarities between MACs and PRFs in the first section of this chapter before reviewing how MACs and PRFs work. Some MACs and PRFs are based on hash functions, some are based on block ciphers, and others are original designs. Finally, we'll discuss examples of attacks on otherwise-secure MACs.

Message Authentication Codes

A MAC protects a message's integrity and authenticity by creating a value $T = \mathbf{MAC}(K, M)$, called the authentication tag of the message, M (often confusingly called the MAC of M). Just as you can decrypt a message if you know a cipher's key, you can validate that a message hasn't been modified if you know a MAC's key.

For example, say Alex and Bill share a key, K, and Alex sends a message, M, to Bill along with its authentication tag, $T = \mathbf{MAC}(K, M)$. Upon receiving the message and its authentication tag, Bill recomputes $\mathbf{MAC}(K, M)$ and checks that it's equal to the authentication tag received. Because only Alex could have computed this value, Bill knows that the message wasn't corrupted in transit (confirming its integrity), whether accidentally or maliciously, and that Alex sent that message (confirming its authenticity).

MACs in Secure Communication

Secure communication systems often combine a cipher and a MAC to protect a message's confidentiality, integrity, and authenticity. For example, the protocols in Internet Protocol Security (IPsec), SSH, and TLS generate a MAC for each transmitted network packet.

Not all communication systems use MACs. Sometimes an authentication tag can add unacceptable overhead to each packet, typically in the range of 64 to 128 bits. For example, the old GSM mobile telephony standard encrypted packets encoding voice calls but didn't authenticate them. An attacker could modify the encrypted audio signal, and the recipient wouldn't notice.

Forgery and Chosen-Message Attacks

What does it mean for a MAC to be secure? First, as with a cipher, the secret key should remain secret. If a MAC is secure, an attacker shouldn't be able to create a tag of some message if they don't know the key. We call a fabricated message/tag pair a *forgery*, and recovering a key is a specific case of a more general class of *forgery attacks*. The security notion that posits that forgeries should be impossible to find is *unforgeability*. It should also be impossible to recover the secret key from a list of tags; otherwise, attackers could forge tags using the key.

What can an attacker do to break a MAC? In other words, what's the attack model? The most basic attack model is the *known-message attack*, which passively collects messages and their associated tags (for example, by eavesdropping on a network). But real attackers can often launch *active* attacks because they can choose the messages to be authenticated and therefore get the MAC of the message they want. The standard model is thus *chosen-message attacks*, wherein attackers get tags for messages of their choice.

Replay Attacks

MACs aren't safe from attacks involving *replays* of tags. For example, if you were to eavesdrop on Alex and Bill's communications, you could capture a message and its tag sent by Alex to Bill and later send them again to Bill pretending to be Alex. To prevent such *replay attacks*, protocols include a message number in each message. This number is incremented for each new message and authenticated along with the message by the MAC. The receiving party gets messages numbered 1, 2, 3, 4, and so on. Thus, if an attacker tries to send message 1 again, the receiver notices that this message is out of order and that it's a potential replay of the earlier message 1.

Pseudorandom Functions

A PRF uses a secret key to return **PRF**(*K*, *M*), such that the output looks random. Because the key is secret, the output values are unpredictable to an attacker.

Unlike MACs, PRFs are not meant to be used on their own but as part of a cryptographic algorithm or protocol. For example, you can use PRFs to create block ciphers within the Feistel construction—see "How to Construct Block Ciphers" in Chapter 4. Key derivation schemes use PRFs to generate cryptographic keys from a master key or password, and identification schemes use PRFs to generate a response from a random challenge. (Basically, a server sends a random challenge message, *M*, and the client returns **PRF**(*K*, *M*) in its response to prove that it knows *K*.) The 5G telephony standard uses a PRF to authenticate a SIM card and its service provider, and a similar PRF generates the encryption key and MAC key to be used during a phone call. The TLS protocol uses a PRF to generate key material from a master secret as well as session-specific random values. There's even a PRF in the noncryptographic `hash()` function built into the Python language to compare objects.

PRF Security

To be secure, a pseudorandom function should have no pattern that sets its outputs apart from truly random values. An attacker who doesn't know the key, *K*, shouldn't be able to distinguish the outputs of **PRF**(*K*, *M*) from random values. Viewed differently, an attacker shouldn't have any means of knowing whether they're talking to a PRF algorithm or to a random function. The erudite phrase for that security notion is " indistinguishability from a random function." (To learn more about the theoretical foundations of PRFs, see Volume 1, Section 3.6 of Goldreich's *Foundations of Cryptography*.)

PRFs Are Stronger Than MACs

PRFs and MACs are keyed hashes, but PRFs are fundamentally stronger than MACs because MACs have weaker security requirements. Whereas you

would consider a MAC secure if an attacker can't forge tags—that is, if they can't guess the MAC's outputs—a PRF is secure only if its outputs are indistinguishable random strings. If you can't distinguish a PRF's outputs from random strings, this implies their values can't be guessed; in other words, any secure PRF is also a secure MAC.

The converse is not true, however: a secure MAC isn't necessarily a secure PRF. For example, say you start with a secure PRF, **PRF1**, and you want to build a second PRF, **PRF2**, from it, like this:

$$\textbf{PRF2}(K, M) = \textbf{PRF1}(K, M) \parallel 0$$

Because **PRF2**'s output is defined as **PRF1**'s output followed by one 0 bit, it doesn't look as random as a true random string, and you can distinguish its outputs by that last 0 bit. Hence, **PRF2** is not a secure PRF. However, because **PRF1** is secure, **PRF2** would still make a secure MAC. If you were able to forge a tag, $T = \textbf{PRF2}(K, M)$, for some M, then you'd also be able to forge a tag for **PRF1**, which you know to be impossible in the first place because PRF1 is a secure MAC. Thus, PRF2 is a keyed hash that's a secure MAC but not a secure PRF.

But don't worry: you won't find such MAC constructions in real applications. In fact, many of the deployed or standardized MACs are also secure PRFs and are often used as either. For example, TLS uses the algorithm HMAC-SHA-256 both as a MAC and as a PRF.

How to Create Keyed Hashes from Unkeyed Hashes

Throughout cryptography's history, MACs and PRFs have rarely been designed from scratch but rather have been built from existing algorithms, usually hash functions or block ciphers. It might seem obvious that you can produce a keyed hash function by feeding an (unkeyed) hash function a key and a message, but this is easier said than done.

The Secret-Prefix Construction

The first technique we'll examine, the *secret-prefix construction*, turns a normal hash function into a keyed hash by prepending the key to the message and returning $\textbf{Hash}(K \parallel M)$. This approach is insecure when the hash function is vulnerable to length-extension attacks (see "The Length-Extension Attack" in Chapter 6) and when the hash supports keys of different lengths.

Insecurity Against Length-Extension Attacks

Recall from Chapter 6 that hash functions of the SHA-2 family allow attackers to compute the hash of a partially unknown message when given a hash of a shorter version of that message. In formal terms, the length-extension attack allows attackers to compute $\textbf{Hash}(K \parallel M_1 \parallel M_2)$ given only $\textbf{Hash}(K \parallel M_1)$ and neither M_1 nor K. These functions allow attackers to forge valid MAC tags for free because they shouldn't be able to guess

the MAC of $M_1 \,||\, M_2$ given only the MAC of M_1. This fact makes the secret-prefix construction insecure as a MAC and PRF when, for example, using it with SHA-256 or SHA-512. It's a weakness of Merkle–Damgård to allow length-extension attacks, and none of the SHA-3 finalists do. The ability to thwart length-extension attacks was mandatory for SHA-3 submissions (see Chapter 6).

Insecurity with Different Key Lengths

The secret-prefix construction is also insecure when allowing the use of keys of different lengths. For example, if the key K is the 24-bit hexadecimal string 123abc and M is def00, then **Hash**() processes the value $K \,||\, M$ = 123abcdef00. If K is instead the 16-bit string 123a and M is bcdef000, then **Hash**() processes $K \,||\, M$ = 123abcdef00, too. Therefore, the result of the secret-prefix construction **Hash**($K \,||\, M$) is the same for both keys.

This problem is independent of the underlying hash; you can fix it by hashing the key's length along with the key and the message—for example, by encoding the key's bit length as a 16-bit integer, L, and then hashing **Hash**($L \,||\, K \,||\, M$). You shouldn't have to do this, however, as modern hash functions such as BLAKE2 and SHA-3 include a keyed mode that avoids these pitfalls and yields a secure PRF and thus a secure MAC as well.

The Secret-Suffix Construction

Instead of hashing the key before the message as in the secret-prefix construction, you can hash it *after*. And that's exactly how the *secret-suffix construction* works by building a PRF from a hash function as **Hash**($M \,||\, K$).

Putting the key at the end makes quite a difference. The length-extension attack that works against secret-prefix MACs won't work against the secret suffix. Applying length extension to a secret-suffix MAC results in **Hash**($M_1 \,||\, K \,||\, M_2$) from **Hash**($M_1 \,||\, K$), but this isn't a valid attack because **Hash**($M_1 \,||\, K \,||\, M_2$) isn't a valid secret-suffix MAC; the key needs to be at the end.

However, the secret-suffix construction is weaker against another type of attack. Say you've got a collision for the hash **Hash**(M_1) = **Hash**(M_2), where M_1 and M_2 are two distinct messages, possibly of different sizes. In the case of an M–D hash function such as SHA-256, this implies that **Hash**($M_1 \,||\, P_1 \,||\, K$) and **Hash**($M_2 \,||\, P_2 \,||\, K$) will be equal too, where P_1 and P_2 are the padding data added to complete a block. After processing $M_1 \,||\, P_1$, the hash function's state is the same as after processing $M_2 \,||\, P_2$. Adding K to each instance preserves the equality of the state, leading to a collision regardless of K's value.

To exploit this property, an attacker:

1. Finds two colliding messages, M_1 and M_2
2. Requests the MAC tag of M_1, **Hash**($M_1 \,||\, K$)
3. Guesses that **Hash**($M_2 \,||\, K$) is the same, thereby forging a valid tag and breaking the MAC's security

The HMAC Construction

The *hash-based MAC (HMAC)* construction allows you to build a MAC from a hash function, which is more secure than either secret prefix or secret suffix. HMAC yields a secure PRF as long as the underlying hash is collision resistant, but even if that isn't the case, HMAC still yields a secure PRF if the hash's compression function is a PRF. The secure communication protocols IPsec, SSH, and TLS have all used HMAC. (You'll find HMAC specifications in the NIST FIPS 198-1 standard and in RFC 2104.)

HMAC uses a hash function, **Hash**, to compute a MAC tag, as Figure 7-1 shows and according to the following expression:

$$\mathbf{Hash}((K \oplus opad) \,||\, \mathbf{Hash}((K \oplus ipad) \,||\, M))$$

The value *opad* (outer padding) is the hexadecimal string `5c5c5c...5c` as long as **Hash**'s block size. The key, *K*, is usually shorter than one block that's filled with `00` bytes and XORed with *opad*. For example, if *K* is the 1-byte string `00`, then $K \oplus opad = opad$. (The same is true if *K* is the all-zero string of any length up to a block's length.) $K \oplus opad$ is the first block processed by the outer call to **Hash**—namely, the leftmost **Hash** in the preceding equation, or the bottom hash in Figure 7-1.

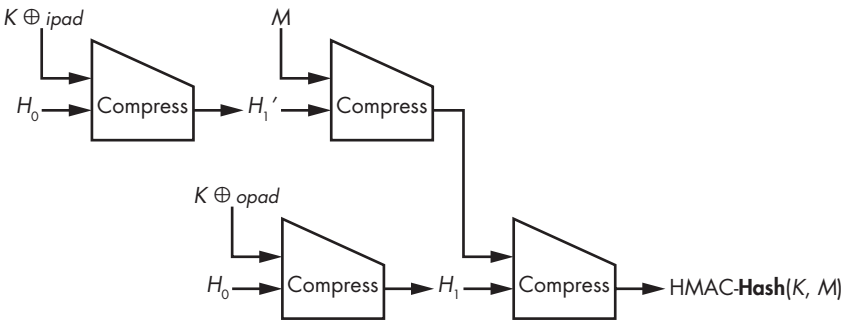


Figure 7-1: The HMAC hash-based MAC construction for a hash function based on a compression function

The value *ipad* (inner padding) is a string (363636 . . . 36) as long as the **Hash**'s block size and completed with 00 bytes. The resulting block is the first processed by the inner call to **Hash**—namely, the rightmost **Hash** in the equation, or the top hash in Figure 7-1.

NOTE *The envelope method is a more secure construction than secret prefix or secret suffix. It's expressed as* ***Hash****(*K || M || K*), a* sandwich MAC*, but it's theoretically less secure than HMAC.*

If SHA-256 is the hash function used as **Hash**, then you call the HMAC instance HMAC-SHA-256. More generally, you call HMAC-*Hash* an HMAC instance using the hash function *Hash*. That means if someone asks you to use HMAC, you should always ask, "Which hash function?"

A Generic Attack Against Hash-Based MACs

There is one attack that works against all MACs based on an iterated hash function. Recall the attack in "The Secret-Suffix Construction" on page 143 where we used a hash collision to get a collision of MACs. You can use the same strategy to attack a secret-prefix MAC or HMAC, though the consequences are less devastating.

Figure 7-2 illustrates the secret-prefix MAC **Hash**($K \,||\, M$).

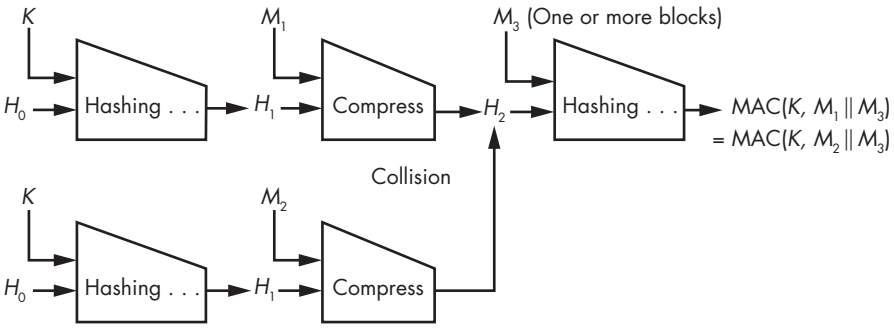


Figure 7-2: The principle of the generic forgery attack on hash-based MACs

If the digest is n bits, you can find two messages, M_1 and M_2, such that **Hash**($K \,||\, M_1$) = **Hash**($K \,||\, M_2$), by requesting approximately $2^{n/2}$ MAC tags to the system holding the key. (Recall the birthday attack from Chapter 6.) If the hash lends itself to length extension, as SHA-256 does, you can then use M_1 and M_2 to forge MACs by choosing some arbitrary data, M_3, and then querying the MAC oracle for **Hash**($K \,||\, M_1 \,||\, M_3$), which is the MAC of message $M_1 \,||\, M_3$. As it turns out, this is also the MAC of message $M_2 \,||\, M_3$ because the hash's internal state of M_1 and M_3 and M_2 and M_3 is the same. You've successfully forged a MAC tag. (The effort becomes infeasible as n grows beyond, say, 128 bits.)

This attack works even if the hash function isn't vulnerable to length extension, and it works for HMAC, too. This is because all that's required is an internal collision in the hash function and not necessarily in the complete hash. The cost of the attack depends on both the size of the chaining value and the MAC's length: if a MAC's chaining value is 512 bits and its tags are 128 bits, a 2^{64} computation would find a MAC collision but probably not a collision in the internal state, since finding such a collision requires $2^{512/2} = 2^{256}$ operations on average.

How to Create Keyed Hashes from Block Ciphers

The compression functions in many hash functions are built on block ciphers (see Chapter 6). For example, HMAC-SHA-256 PRF is a series of calls to SHA-256's compression function, which is a block cipher that repeats a sequence of rounds. In other words, HMAC-SHA-256 is a block cipher inside a compression function inside a hash inside the HMAC construction. So why not use a block cipher directly rather than build such a layered construction?

Cipher-based MAC (CMAC) is such a construction: it creates a MAC given only a block cipher, such as AES. Though less popular than HMAC, CMAC is deployed in many systems, including the Internet Key Exchange (IKE) protocol, which is part of the IPsec suite. IKE, for example, generates key material using the AES-CMAC-PRF-128 construction as a core algorithm (or CMAC based on AES with 128-bit output). CMAC is specified in RFC 4493.

Breaking CBC-MAC

CMAC was designed in 2005 as an improved version of *CBC-MAC*, a simpler block cipher–based MAC derived from the cipher block chaining (CBC) block cipher mode of operation (see "Modes of Operation" in Chapter 4).

CBC-MAC, the ancestor of CMAC, is simple: to compute the tag of a message, M, given a block cipher, $\mathbf{E}$, encrypt M in CBC mode with an all-zero initial value (IV) and discard all but the last ciphertext block. That is, compute

$$C_1 = \mathbf{E}(K, M_1),\ C_2 = \mathbf{E}(K, M_2 \oplus C_1),\ C_3 = \mathbf{E}(K, M_3 \oplus C_2)$$

and so on for each of M's blocks and keep only the last C_i—your CBC-MAC tag for M—simple, and simple to attack.

To understand why CBC-MAC is insecure, consider the CBC-MAC tag, $T_1 = \mathbf{E}(K, M_1)$, of a single-block message, M_1, and the tag, $T_2 = \mathbf{E}(K, M_2)$, of another single-block message, M_2. Given these two pairs, (M_1, T_1) and (M_2, T_2), you can deduce that T_2 is also the tag of the two-block message $M_1 \,||\, (M_2 \oplus T_1)$. If you apply CBC-MAC to $M_1 \,||\, (M_2 \oplus T_1)$ and compute $C_1 = \mathbf{E}(K, M_1) = T_1$ followed by $C_2 = \mathbf{E}(K, (M_2 \oplus T_1) \oplus T_1) = \mathbf{E}(K, M_2) = T_2$, you can create a third message/tag pair from two message/tag pairs without knowing the key. That is, you can forge CBC-MAC tags, thereby breaking CBC-MAC's security.

Fixing CBC-MAC

CMAC fixes CBC-MAC by processing the last block using a different key from the preceding blocks. To do this, CMAC first derives two keys, K_1 and K_2, from the main key, K, such that K, K_1, and K_2 are distinct. CMAC processes the last block using either K_1 or K_2, while the preceding blocks use K.

To determine K_1 and K_2, CMAC first computes a temporary value, $L = \mathbf{E}(0, K)$, where 0 acts as the key of the block cipher and K acts as the

plaintext block. Then CMAC sets K_1 to $(L \ll 1)$ if L's most significant bit (MSB) is 0 and to $(L \ll 1) \oplus 87$ if L's MSB is 1. (The hexadecimal number 87, or 135 in decimal, is chosen for its mathematical properties when data blocks are 128 bits; a different value is needed when blocks aren't 128 bits.) The key K_2 is set to $(K_1 \ll 1)$ if K_1's MSB is 0 and to $K_2 = (K_1 \ll 1) \oplus 87$ otherwise.

Given K_1 and K_2, CMAC works like CBC-MAC, except for the last block. If the final message chunk M_n is exactly the size of a block, CMAC returns $\mathbf{E}(K, M_n \oplus C_{n-1} \oplus K_1)$ as a tag, as Figure 7-3 illustrates.

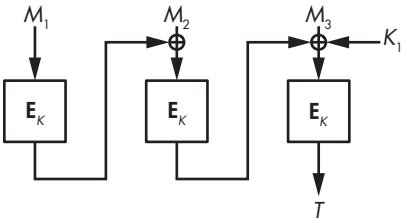


Figure 7-3: The CMAC block cipher–based MAC construction when the message is a sequence of integral blocks

If M_n has fewer bits than a block, CMAC pads it with a 1 bit and 0s and returns $\mathbf{E}(K, M_n \oplus C_{n-1} \oplus K_2)$ as a tag, as in Figure 7-4.

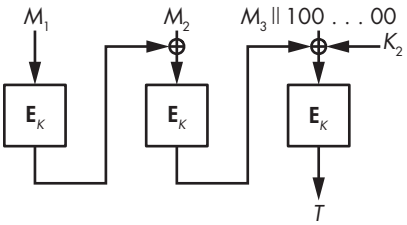


Figure 7-4: The CMAC block cipher–based MAC construction when the last block of the message has to be padded with 1 bit and zeros to fill a block

The first case uses only K_1 and the second only K_2, but both use the main key K to process the message chunks that precede the final one.

Unlike the CBC encryption mode, CMAC doesn't take an IV as a parameter and is deterministic: for the same key, CMAC always returns the same tag for a given message, M, because the computation of **CMAC**(M) isn't randomized—and that's fine because unlike encryption, MAC computation doesn't have to be randomized to be secure, which eliminates the burden of having to choose random IV.

Dedicated MAC Designs

You've seen how to recycle hash functions and block ciphers to build PRFs that are secure as long as their underlying hash or cipher is secure. Schemes such as HMAC and CMAC simply combine available hash functions or block ciphers to yield a secure PRF or MAC. Reusing available algorithms is convenient, but is it the most efficient approach?

Intuitively, PRFs and MACs should require less work to be secure than unkeyed hash functions—their use of a secret key prevents attackers from playing with the algorithm because they don't have the key. Also, PRFs and MACs expose only a short tag to attackers, unlike block ciphers, which expose a ciphertext as long as the message. Hence, PRFs and MACs shouldn't need the maximum power of hash functions or block ciphers—this is the point of *dedicated design*, that is, algorithms created solely to serve as PRFs or MACs.

The sections that follow focus on two such algorithms: Poly1305 and SipHash. I'll explain their design principles and why they're (most likely) secure.

Poly1305

The Poly1305 algorithm (pronounced *poly-thirteen-o-five*) was designed in 2005 by Daniel J. Bernstein (creator of the Salsa20 stream cipher from Chapter 5 and the ChaCha cipher that inspired the BLAKE and BLAKE2 hash functions from Chapter 6). Poly1305 is optimized to be superfast on modern CPUs, and as I write this, it's one of the algorithms supported in TLS 1.3 and in OpenSSH, among many other applications. Unlike Salsa20, the design of Poly1305 is built on techniques dating back to the 1970s—namely, universal hash functions and the Wegman–Carter construction.

Universal Hash Functions

The Poly1305 MAC uses a *universal hash function*. Such a hash function is weaker than a cryptographic hash function but much faster. Universal hash functions don't have to be collision resistant, for example.

Like a PRF, a universal hash is parameterized by a secret key: given a message, M, and key, K, we'll write $\mathbf{UH}(K, M)$ for the output of a universal hash function denoted **UH**. A universal hash function has only one security requirement: for any two messages M_1 and M_2, and a random key K, the probability that $\mathbf{UH}(K, M_1)$ equals $\mathbf{UH}(K, M_2)$ must be negligible. Unlike a PRF, a universal hash doesn't need to be pseudorandom; there simply should be no pair (M_1, M_2) that gives the same hash for many different keys. Because their security requirements are easier to satisfy, universal hash functions require fewer operations and are considerably faster than PRFs.

You can use a universal hash as a MAC to authenticate no more than one message, however. For example, consider the universal *polynomial-evaluation* hash used in Poly1305. (See the seminal 1974 article "Codes Which Detect Deception" by Edgar Gilbert, Jessie MacWilliams, and Neil

Sloane for more on this notion.) This kind of polynomial-evaluation hash is parameterized by a prime number, p, and takes as input a key consisting of two numbers, R and K, in the range $[1, p)$ and a message, M, consisting of n blocks $(M_1, M_2, \ldots, M_n)$. You then compute the output of the universal hash as follows:

$$\mathbf{UH}(R, K, M) = R + M_1K + M_2K^2 + M_3K^3 + \ \ldots \ + M_nK^n \bmod p$$

In this equation, the plus sign (+) denotes the addition of positive integers, K^i is the number K raised to the power i, and "mod p" denotes the reduction modulo p of the result (that is, the remainder of the division of the result by p; for example, 12 mod 10 = 2, 10 mod 10 = 0, 8 mod 10 = 8, and so on).

Because you want the hashing operation to be as fast as possible, universal hash-based MACs often work with message blocks of 128 bits and with a prime number, p, that's slightly larger than 2^{128}, such as $2^{128} + 51$. The 128-bit width allows for very fast implementations by efficiently using the 32- and 64-bit arithmetic units of common CPUs.

Security Limitations

Universal hashes have one weakness: because a universal hash securely authenticates only one message, an attacker could break the preceding polynomial-evaluation MAC by requesting the tags of only two messages. Specifically, they could request the tag for a message where all blocks are zero—$M_1 = M_2 = \ldots = 0$—to obtain the tag $\mathbf{UH}(R, K, 0) = R$, thus finding the secret value R. Alternatively, they could request the tags for a message where $M_1 = 1$ and where $M_2 = M_3 = \ldots = 0$ (whose tag is $T = R + K$), which allows them to find K by subtracting R from T. Now the attacker knows the whole key (R, K) and can forge MACs for any message.

Fortunately, there's a way to go from single-message security to multi-message security.

Wegman–Carter MACs

The trick to authenticating multiple messages using a universal hash function arrived thanks to IBM researchers Mark Wegman and Lawrence Carter and their 1981 paper "New Hash Functions and Their Use in Authentication and Set Equality." The Wegman–Carter construction builds a MAC from a universal hash function and a PRF, using two keys, K_1 and K_2, and it returns

$$\mathbf{MAC}(K_1, K_2, N, M) = \mathbf{UH}(K_1, M) + \mathbf{PRF}(K_2, N)$$

where N is a nonce that must be used at most once for each key, K_2, and where **PRF**'s output is as large as that of the universal hash function **UH**. By adding these two values, **PRF**'s strong pseudorandom output masks the cryptographic weakness of **UH**. You can see this as the encryption of the universal hash's result, where the PRF acts as a stream cipher and prevents

the preceding attack by making it possible to authenticate multiple messages with the same key, K_1.

To recap, the Wegman–Carter construction $\mathbf{UH}(K_1, M) + \mathbf{PRF}(K_2, N)$ gives a secure MAC if you assume the following:

- **UH** is a secure universal hash.
- **PRF** is a secure PRF.
- Each nonce N is used only once for each key K_2.
- The output values of **UH** and **PRF** are long enough to ensure high enough security.

Now let's see how Poly1305 leverages the Wegman–Carter construction to build a secure and fast MAC.

Poly1305-AES

Poly1305 was initially proposed as Poly1305-AES, combining the Poly1305 universal hash with the AES block cipher. Poly1305-AES is much faster than HMAC-based MACs, or even than CMACs, since it computes only one block of AES and processes the message in parallel through a series of simple arithmetic operations.

Given 128-bit K_1, K_2, and N and the message M, Poly1305-AES returns the following:

$$\mathbf{Poly1305}\left(K_1, M\right) + \mathbf{AES}\left(K_2, N\right) \bmod 2^{128}$$

The mod 2^{128} reduction ensures that the result fits in 128 bits. Poly1305 parses the message M as a sequence of 128-bit blocks $(M_1, M_2, \ldots, M_n)$ and appends a 129th bit to each block's most significant bit to make all blocks 129 bits long. (If the last block is smaller than 16 bytes, it's padded with a 1 bit followed by 0 bits before the final 129th bit.) Next, Poly1305 evaluates the polynomial to compute the following:

$$\mathbf{Poly1305}\left(K_1, M\right) = M_1 K_1^{\,n} + M_2 K_1^{\,n-1} + \ \ldots\ + M_n K_1 \bmod 2^{130} - 5$$

The result of this expression is an integer that's at most 129 bits long. When you add this to the 128-bit value $\mathbf{AES}(K_2, N)$, the result is reduced modulo 2^{128} to produce a 128-bit MAC.

NOTE *I described the Wegman–Carter as using a PRF, but AES isn't a PRF, it's a pseudorandom permutation (PRP). However, that doesn't matter because the Wegman–Carter construction works well with PRPs and PRFs. This is because if you're given a function that's either a PRF or a PRP, it's hard to determine whether it's a PRF of a PRP just by looking at the function's output values. In other words, distinguishing a PRP from a PRF is computationally hard.*

The security analysis of Poly1305-AES (see "The Poly1305-AES Message-Authentication Code" at *https://cr.yp.to/mac/poly1305-20050329.pdf*) shows

that Poly1305-AES is 128-bit secure as long as AES is a secure block cipher—and everything is implemented correctly, as with any cryptographic algorithm.

You can combine the Poly1305 universal hash with algorithms other than AES. For example, Poly1305 was used with the stream cipher ChaCha (see RFC 7539, "ChaCha20 and Poly1305 for IETF Protocols"). There's no doubt that Poly1305 will continue being used wherever a fast MAC is needed.

Although Poly1305 is fast and secure, it has several downsides. For one, its polynomial evaluation is difficult to implement efficiently, especially in the hands of those unfamiliar with the associated mathematical notions. (See examples at *https://github.com/floodyberry/poly1305-donna*.) Second, on its own, it's secure for only one message unless you use the Wegman–Carter construction. But in that case it requires a nonce, and if the nonce is repeated, the algorithm becomes insecure. Finally, Poly1305 is optimized for long messages, but it's overkill if you process only small messages (say, fewer than 128 bytes). In such cases, SipHash is a good solution.

SipHash

I designed SipHash in 2012 with Dan Bernstein initially to address a noncryptographic problem: denial-of-service attacks on hash tables. Hash tables are data structures, notably used in programming languages to efficiently store elements internally. Prior to SipHash, hash tables relied on noncryptographic keyed hash functions for which collisions were often easy to find, which could be exploited to slow down the system and perform denial-of-service attacks. We observed that a PRF would address this problem and set out to design SipHash, a PRF suitable for hash tables. Because hash tables process mostly short inputs, SipHash is optimized for short messages. SipHash is a full-blown PRF and MAC that shines where most inputs are short.

SipHash uses a trick that makes it more secure than basic sponge functions: instead of XORing message blocks only once before the permutation, SipHash XORs them before and after the permutation, as Figure 7-5 illustrates. The 128-bit key of SipHash is seen as two 64-bit words, K_1 and K_2, XORed to a 256-bit fixed initial state that's seen as four 64-bit words. Next, the keys are discarded, and computing SipHash boils down to iterating through a core function called SipRound and then XORing message chunks to modify the four-word internal state. Finally, SipHash returns a 64-bit tag by XORing the four-state words together.

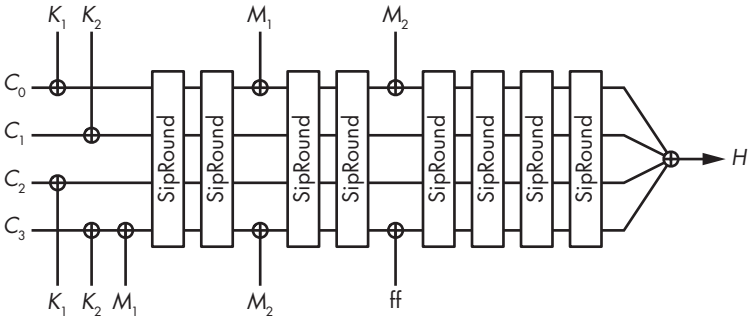


Figure 7-5: SipHash-2-4 processing a 15-byte message (a block, M_1, of 8 bytes and a block, M_2, of 7 bytes, plus 1 byte of padding)

The SipRound function uses a bunch of XORs together with additions and word rotations to make the function secure. SipRound transforms a state of four 64-bit words (*a*, *b*, *c*, *d*) by performing the following operations, top to bottom. The operations on the left and on the right are independent and can be carried out in parallel:

$a \mathrel{+}= b$	$c \mathrel{+}= d$
$b \lll= 13$	$d \lll= 16$
$b \mathrel{\oplus}= a$	$d \mathrel{\oplus}= c$
$a \lll= 32$	
$c \mathrel{+}= b$	$a \mathrel{+}= d$
$b \lll= 17$	$d \lll= 21$
$b \mathrel{\oplus}= c$	$d \mathrel{\oplus}= a$
$c \lll= 32$	

Here, $a \mathrel{+}= b$ is shorthand for $a = a + b$, and $b \lll= 13$ is shorthand for $b = b \lll 13$ (the 64-bit word *b* left-rotated 13 bits).

These simple operations on 64-bit words are almost all you need to implement to compute SipHash—although you won't have to implement it yourself. You can find readily available implementations in most languages, such as C, Go, Java, JavaScript, Python, and Rust.

NOTE *I wrote* ***SipHash***-x-y *as the SipHash version, meaning it makes* x *SipRounds between each message block injection and then* y *rounds. More rounds require more operations, which slows down operations but also increases security. The default version is SipHash-2-4 (simply noted as SipHash), and it has so far resisted cryptanalysis. Note that I also defined SipHash128, a version of SipHash producing 128-bit tags.*

Many systems, such as the Rust language, the OpenBSD operating system, and the Bitcoin blockchain, use SipHash internally. The Linux kernel uses SipHash too and also uses HalfSipHash, "SipHash's insecure younger cousin," a smaller version with a 64-bit key and 32-bit output (see *https://docs.kernel.org/security/siphash.html*).

How Things Can Go Wrong

Like ciphers and unkeyed hash functions, MACs and PRFs that are secure on paper can be vulnerable to attacks when used in a real setting. Let's discuss two examples.

Timing Attacks on MAC Verification

Side-channel attacks target the implementation of a cryptographic algorithm rather than the algorithm itself. In particular, *timing attacks* use an algorithm's execution time to determine secret information, such as keys, plaintext, and secret random values. Variable-time string comparison induces vulnerabilities not only in MAC verification but also in many other cryptographic and security functionalities.

MACs can be vulnerable to timing attacks when a remote system verifies tags in a period of time that depends on the tag's value, thereby allowing an attacker to determine the correct message tag by trying many incorrect ones to determine the one that takes the longest amount of time to complete. The problem occurs when a server compares the correct tag with an incorrect one by comparing the two strings byte per byte, in order, until the bytes differ. For example, the Python code in Listing 7-1 compares two strings byte per byte, in variable time: if the first bytes differ, the function returns after only one comparison; if the strings `x` and `y` are identical, the function makes `n` comparisons against the length of the strings.

```
def compare_mac(x, y, n):
    if len(x) != len(y):
        return False
    if len(x) != n:
        return False
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True
```

Listing 7-1: A comparison of two `n`-byte strings, taking variable time

To demonstrate the vulnerability of the `compare_mac()` function, we'll write a program that measures the execution time of 100,000 calls to `compare_mac()`, first with identical 16-character `x` and `y` values and then with `x` and `y` values that differ in their third byte. The latter comparison should take noticeably less time because `compare_mac()` compares fewer bytes than the identical `x` and `y`, as Listing 7-2 shows.

```
from time import time

MAC1 = 'abcdefghijklmnop'
MAC2 = 'abXdefghijklmnop'
TRIALS = 100000

def compare_mac(x, y, n):
    if len(x) != len(y):
        return False
```

```
    if len(x) != n:
        return False
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True

# Each call to compare_mac() will look at all 16 characters.
start = time()
for i in range(TRIALS):
    compare_mac(MAC1, MAC1, len(MAC1))
end = time()
print("%0.5f" % (end-start))

# Each call to compare_mac() will look at three characters.
start = time()
for i in range(TRIALS):
    compare_mac(MAC1, MAC2, len(MAC1))
end = time()
print("%0.5f" % (end-start))
```

Listing 7-2: Measuring timing differences when executing `compare_mac()` from Listing 7-1

On a MacBook Pro with an ARM M1 chip, an execution of the program in Listing 7-2 prints execution times of around 67 and 26 milliseconds, respectively. That difference is significant enough to identify what's happening within the algorithm. Now move the difference to other offsets in the string, and you'll observe different execution times for different offsets. If `MAC1` is the correct MAC tag and `MAC2` is the one tried by the attacker, you can easily identify the position of the first difference, which is the number of correctly guessed bytes.

If execution time doesn't depend on a secret timing, timing attacks won't work, which is why implementers strive to write *constant-time* implementations—that is, code that takes exactly the same time to complete for any secret input value. For example, the C function in Listing 7-3 compares two buffers of `size` bytes in constant time: the temporary variable `result` is nonzero if and only if there's a difference somewhere in the two buffers.

```
int cmp_const(const void *a, const void *b, const size_t size)
{
    const unsigned char *_a = (const unsigned char *) a;
    const unsigned char *_b = (const unsigned char *) b;
    unsigned char result = 0;
    size_t i;

    for (i = 0; i < size; i++) {
        result |= _a[i] ^ _b[i];
    }

    return result; /* Returns 0 if *a and *b are equal, nonzero otherwise */
}
```

Listing 7-3: A constant-time comparison of two buffers for safer MAC verification

When Sponges Leak

Permutation-based algorithms like SHA-3 and SipHash are simple, are easy to implement, and come with compact implementations, but they're fragile in the face of side-channel attacks that recover a snapshot of the system's state. For example, if a process can read the RAM and registers' values at any time or read a core dump of the memory, an attacker can determine the internal state of SHA-3 in MAC mode, or the internal state of SipHash, and then compute the reverse of the permutation to recover the initial secret state. They can then forge tags for any message, breaking the MAC's security.

Fortunately, this attack won't work against compression function–based MACs such as HMAC-SHA-256 and keyed BLAKE2 because the attacker requires a snapshot of memory at the exact time when the key is used. The upshot is that if you're in an environment where parts of a process's memory may leak, you can use a MAC based on a noninvertible transform compression function rather than on a permutation.

Further Reading

The venerable HMAC deserves more attention than I have space for here, and even more for the train of thought that led to its wide adoption and eventual demise when combined with a weak hash function. I recommend the 1996 paper "Keying Hash Functions for Message Authentication" by Mihir Bellare, Ran Canetti, and Hugo Krawczyk, which introduced HMAC and its cousin NMAC, and the 2006 follow-up paper by Bellare called "New Proofs for NMAC and HMAC: Security Without Collision-Resistance," which proves that HMAC doesn't need a collision-resistant hash but only a hash with a compression function that is a PRF. On the offensive side, the 2007 paper "Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5" by Pierre-Alain Fouque, Gaëtan Leurent, and Phong Nguyen shows how to attack HMAC and NMAC when they're built on top of a brittle hash function such as MD4 or MD5. (HMAC-MD5 and HMAC-SHA-1 aren't totally broken, but the risk is high enough.)

The Wegman–Carter MACs are also worth more attention, both for their practical interest and for their underlying theory. The seminal papers by Wegman and Carter are available at *https://cr.yp.to/bib/entries.html*. Other state-of-the-art designs include UMAC and VMAC, which are among the fastest MACs on long messages.

One type of MAC not discussed in this chapter is *Pelican*, which uses the AES block cipher reduced to four rounds (down from 10 in the full block cipher) to authenticate chunks of messages within a simplistic construction, as described in *https://eprint.iacr.org/2005/088*. Pelican is more of a curiosity, though, and it's rarely used in practice.

Last but not least, if you're interested in finding vulnerabilities in cryptographic software, look for uses of CBC-MAC or for weaknesses caused by HMAC handling keys of arbitrary sizes—taking **Hash**(K) as the key

rather than K if K is too long, thus making K and **Hash**(K) *equivalent keys*. Or just look for systems that don't use MAC when they should—a frequent occurrence.

In Chapter 8, we'll combine MACs with ciphers to protect a message's authenticity, integrity, *and* confidentiality. We'll also do this without MACs, thanks to authenticated ciphers, which combine the functionality of a basic cipher with that of a MAC by returning a tag along with each ciphertext.

8

AUTHENTICATED ENCRYPTION

This chapter is about a type of algorithm that protects not only a message's confidentiality but also its authenticity. Recall from Chapter 7 that message authentication codes (MACs) protect a message's authenticity by creating a tag, which is a kind of signature. Like MACs, the authenticated encryption (AE) algorithms in this chapter produce an authentication tag, but they also encrypt the message. In other words, a single AE algorithm offers the features of both a normal cipher and a MAC.

Combining a cipher and a MAC can achieve varying levels of authenticated encryption, as you'll learn throughout this chapter. We'll review several ways to combine MACs with ciphers, discuss which methods are the most secure, and explore ciphers that produce both a ciphertext and an authentication tag. We'll then look at four important authenticated

ciphers: three block cipher–based constructions, with a focus on the popular Advanced Encryption Standard in Galois Counter Mode (AES-GCM), and a cipher that uses only a permutation algorithm.

Authenticated Encryption Using MACs

Figure 8-1 shows three ways that MACs and ciphers can be combined to both encrypt and authenticate a plaintext: encrypt-and-MAC, MAC-then-encrypt, and encrypt-then-MAC.

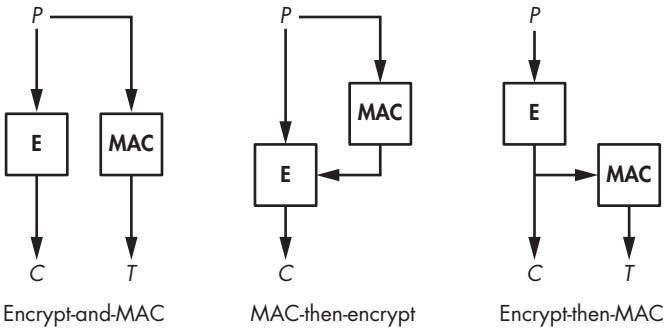


Figure 8-1: Cipher and MAC combinations

These combinations differ in the order in which you apply encryption and generate the authentication tag. The choice of a specific MAC or cipher algorithm is unimportant as long as each is secure in its own right and the MAC and cipher use distinct keys.

In the encrypt-and-MAC composition, the plaintext is encrypted and an authentication tag is generated directly from the plaintext, such that the two operations (encryption and authentication) are independent of each other and you can therefore compute them in parallel. In the MAC-then-encrypt scheme, you generate the tag from the plaintext first and then encrypt the plaintext and MAC together. In the encrypt-then-MAC method, you encrypt the plaintext first and then generate the tag from the ciphertext. Let's see which method is likely to be the most secure.

Encrypt-and-MAC Approach

The *encrypt-and-MAC* approach computes a ciphertext and a MAC tag separately. Given a plaintext P, the sender computes a ciphertext $C = \mathbf{E}(K_1, P)$, where **E** is an encryption algorithm and C is the resulting ciphertext. You calculate the authentication tag T from the plaintext as $T = \mathbf{MAC}(K_2, P)$. The two operations are independent and can therefore be computed in parallel.

Once you've generated the ciphertext and authentication tag, the sender transmits both to the intended recipient. When the recipient receives C and T, they decrypt C to obtain the plaintext P by computing $P = \mathbf{D}(K_1, C)$. Next, they compute $\mathbf{MAC}(K_2, P)$ using the decrypted plaintext and compare the

result to the T received. This verification fails if either C or T was corrupted, and the message is deemed invalid.

In theory, encrypt-and-MAC is the least secure MAC and cipher composition because even a secure MAC could leak information on P, making P easier to recover. Because the goal of using MACs is simply to make tags unforgeable and because tags aren't necessarily random looking, the authentication tag (T) of a plaintext (P) could still leak information even though the MAC is considered secure! (If the MAC is a pseudorandom function, the tag won't leak anything on P.)

Still, despite its relative weakness, many systems continue supporting encrypt-and-MAC, including the secure transport layer protocol SSH, wherein each encrypted packet C is followed by the tag $T = \mathbf{MAC}(K, N \| P)$, where N is a 32-bit sequence number that increments for each packet. In practice, encrypt-and-MAC has proven good enough for use with SSH, thanks to the use of strong MAC algorithms like HMAC-SHA-256 that don't leak information on P. Enter the following command

```
$ ssh -Q mac
```

to see the list of MACs supported by the OpenSSH software.

MAC-Then-Encrypt Composition

The *MAC-then-encrypt* composition protects a message, P, by first computing the authentication tag $T = \mathbf{MAC}(K_2, P)$. Next, it creates the ciphertext by encrypting the plaintext and tag together, according to $C = \mathbf{E}(K_1, P \| T)$.

Once these steps are complete, the sender transmits only C, which contains both the encrypted plaintext and tag. Upon receipt, the recipient decrypts C by computing $P \| T = \mathbf{D}(K_1, C)$ to obtain the plaintext and tag T. Next, the recipient verifies the received tag T by computing a tag directly from the plaintext according to $\mathbf{MAC}(K_2, P)$ to confirm that the computed tag is equal to the tag T.

As with encrypt-and-MAC, when using MAC-then-encrypt, the recipient must decrypt C before determining whether they're receiving corrupted packets—a process that exposes potentially corrupted plaintexts to the receiver. Nevertheless, MAC-then-encrypt is more secure than encrypt-and-MAC because it hides the plaintext's authentication tag, thus preventing the tag from leaking information on the plaintext.

The TLS protocol has used MAC-then-encrypt for years, but TLS 1.3 replaced MAC-then-encrypt with authenticated ciphers (see Chapter 13 for more on TLS 1.3).

Encrypt-Then-MAC Composition

The encrypt-then-MAC composition sends two values to the recipient: the ciphertext produced by $C = \mathbf{E}(K_1, P)$ and a tag based on the ciphertext, $T = \mathbf{MAC}(K_2, C)$. The receiver computes the tag using $\mathbf{MAC}(K_2, C)$ and verifies that it equals the T received. If the values are equal, the plaintext is computed as $P = \mathbf{D}(K_1, C)$; if they are not equal, the ciphertext is discarded.

One advantage of this method is that the receiver needs to compute a MAC only to detect corrupt messages, meaning that there's no need to decrypt a corrupt ciphertext. Also, attackers can't send pairs of C and T to the receiver to decrypt unless they've broken the MAC, which makes it harder for attackers to transmit malicious data to the recipient.

This combination of features makes encrypt-then-MAC stronger than the encrypt-and-MAC and MAC-then-encrypt approaches. This is one reason why the widely used IPsec secure communications protocol suite uses it to protect packets (for example, within VPN tunnels).

Note that SSH and TLS don't use encrypt-then-MAC because other approaches appeared adequate when SSH and TLS were created—not because theoretical weaknesses didn't exist but because undesirable properties don't necessarily become actual vulnerabilities.

Authenticated Ciphers

Authenticated ciphers are an alternative to the cipher and MAC combinations. They're like normal ciphers except that they return an authentication tag together with the ciphertext.

You represent the authenticated cipher encryption as $\mathbf{AE}(K, P) = (C, T)$. The term **AE** stands for *authenticated encryption*, which is based on a key (K) and a plaintext (P) and returns a ciphertext (C) and a generated authentication tag (T). In other words, a single authenticated cipher algorithm does the same job as a cipher and MAC combination, making it simpler, faster, and often more secure.

You represent authenticated cipher decryption by $\mathbf{AD}(K, C, T) = P$. Here, **AD** stands for *authenticated decryption*, which returns a plaintext (P) given a ciphertext (C), tag (T), and key (K). If the tag validation fails, **AD** returns an error to prevent the recipient from processing a plaintext that may have been forged. By the same token, if **AD** returns a plaintext, it's been encrypted by someone or something that knows the secret key.

The basic security requirements of an authenticated cipher are simple: its authentication should be as strong as a MAC's, meaning it should be impossible to forge a ciphertext and tag pair (C, T) that the decryption function **AD** will accept and decrypt.

As far as confidentiality is concerned, an authenticated cipher is fundamentally stronger than a basic cipher because systems holding the secret key will decrypt a ciphertext only if the authentication tag is valid. If the tag is invalid, the plaintext is discarded. This characteristic prevents attackers from performing chosen-ciphertext queries, an attack where they create ciphertexts and ask for the corresponding plaintext.

Authenticated Encryption with Associated Data

Cryptographers define *associated data* as any data processed by an authenticated cipher such that the data is authenticated (thanks to the authentication tag) but not encrypted. By default, all plaintext data fed to an authenticated cipher is encrypted *and* authenticated.

Say you want to authenticate a message, including its unencrypted parts, but not encrypt the entire message—that is, you want to authenticate and transmit data in addition to an encrypted message. For example, if a cipher processes a network packet composed of a header followed by a payload, you might choose to encrypt the payload to hide the actual data transmitted, but not encrypt the header since it contains information required to deliver the packet to its final recipient. At the same time, you might still like to authenticate the header's data to make sure that it's received from the expected sender.

To accomplish these goals, cryptographers created the notion of *authenticated encryption with associated data (AEAD)*. An AEAD algorithm allows you to attach cleartext data to a ciphertext in such a way that if the cleartext data is corrupted, the authentication tag won't validate, and the ciphertext won't decrypt. Such cleartext data must be encoded and serialized in a secure way to prevent ambiguous interpretation of its content.

You can write an AEAD operation as **AEAD**(K, P, A) = (C, A, T). Given a key (K), plaintext (P), and associated data (A), AEAD returns the ciphertext, the unencrypted associated data A, and an authentication tag. AEAD leaves the unencrypted associated data unchanged, and the ciphertext is the encryption of plaintext. The authentication tag depends on both P and A and will be verified as valid only if neither C nor A has been modified.

Because the authenticated tag depends on A, you compute decryption with associated data by **ADAD**(K, C, A, T) = (P, A). Decryption requires the key, ciphertext, associated data, and tag in order to compute the plaintext and associated data, and it fails if either C or A has been corrupted.

When using AEAD, you can leave A or P empty. If the associated data A is empty, AEAD becomes a normal authenticated cipher; if P is empty, it's just a MAC.

NOTE *As of this writing, AEAD is the current norm for authenticated encryption. Because nearly all authenticated ciphers in use today support associated data, when referring to authenticated ciphers throughout this book, I'm referring to AEAD unless stated otherwise. When discussing AEAD operations of encryption and decryption, I'll refer to them as* ***AE*** *and* ***AD****, respectively.*

Predictability and Nonces

Recall from Chapter 1 that to be secure, encryption schemes must be unpredictable and return different ciphertexts when called repeatedly to encrypt the same plaintext—otherwise, an attacker can determine whether the same plaintext was encrypted twice. To be unpredictable, block ciphers and stream ciphers feed the cipher an extra parameter: the initial value (IV) or *nonce*—a number that can be used only once. Authenticated ciphers use the same trick. Thus, you express authenticated encryption as **AE**(K, P, A, N), where N is a nonce. It's up to the encryption operation to pick a nonce that has never been used before with the same key.

As with block and stream ciphers, decryption with an authenticated cipher requires the nonce used for encryption to perform correctly. You

can thus express decryption as $\mathbf{AD}(K, C, A, T, N) = (P, A)$, where N is the nonce used to create C and T.

Criteria for a Good Authenticated Cipher

Researchers have been struggling since the early 2000s to define what makes a good authenticated cipher, and the answer is still elusive. Because of AEAD's many inputs that play different roles, it's harder to define a notion of security than it is for basic ciphers that only encrypt a message. For example, the research article "Nonces Are Noticed: AEAD Revisited," available at *https://eprint.iacr.org/2019/624*, proposed a theoretical framework for nonce-based encryption. Nevertheless, in this section, I'll summarize the most important criteria to consider when evaluating the security, performance, and functionality of an authenticated cipher.

Security

The most important criteria to measure the strength of an authenticated cipher are its ability to protect the confidentiality of data (that is, the secrecy of the plaintext) and the authenticity and integrity of the communication (as with the MAC's ability to detect corrupted messages). An authenticated cipher must compete in both leagues: its confidentiality must be as strong as that of the strongest cipher, and its authenticity as strong as that of the best MAC. In other words, if you remove the authentication part in an AEAD, you should get a secure cipher, and if you remove the encryption part, you should get a strong MAC.

Another measure of the strength of an authenticated cipher's security is based on its fragility when faced with repeated nonces. For example, if a nonce is reused, can an attacker decrypt ciphertexts or learn the difference between plaintexts?

Researchers call this notion of robustness *misuse resistance* and have designed misuse-resistant authenticated ciphers to weigh the impact of a repeated nonce and attempt to determine whether confidentiality, authenticity, or both would be compromised in the face of such an attack, as well as what information about the encrypted data would likely leak.

Performance

As with every cryptographic algorithm, you measure the throughput of an authenticated cipher in bits processed per second. This speed depends on the number of operations performed by the cipher's algorithm and on the extra cost of the authentication functionality. The extra security features of authenticated ciphers come with a performance hit. However, the measure of a cipher's performance isn't just about pure speed. It's also about parallelizability, structure, and whether the cipher is streamable. Let's examine these notions more closely.

A cipher's *parallelizability* is a measure of its ability to process multiple data blocks simultaneously without waiting for the previous block's

processing to complete. Block cipher–based designs can be easily parallelizable when each block can be processed independently of the other blocks. For example, the CTR block cipher mode from Chapter 4 is parallelizable, whereas the CBC encryption mode is not, because blocks are chained.

The internal structure of an authenticated cipher is another important performance criteria. There are two main types of structure: one-layer and two-layer. In a two-layer structure (for example, in the widely used AES-GCM), one algorithm processes the plaintext, and then a second algorithm processes the result. Typically, the first layer is the encryption layer, and the second is the authentication layer. But as you might expect, a two-layer structure complicates implementation and tends to slow down computations.

An authenticated cipher is *streamable* (also called an *online* cipher) when it can process a message block by block and discard any already-processed blocks. In contrast, nonstreamable ciphers must store the entire message, typically because they need to make two consecutive passes over the data: one from the start to the end and the other from the end to the start of the data obtained from the first pass.

Because of potentially high memory requirements, some applications won't work with nonstreamable ciphers. For example, a router could receive an encrypted block of data, decrypt it, and then return the plaintext block before moving on to decrypt the subsequent block of the message, though the recipient of the decrypted message would still have to verify the authentication tag sent at the end of the decrypted data stream.

Other Features

Functional criteria are the features of a cipher or its implementation that don't directly relate to either security or performance. For example, some authenticated ciphers allow only associated data to precede the data to be encrypted (because they need access to it to start encryption). Others require associated data to follow the data to be encrypted or support the inclusion of associated data anywhere—even between chunks of plaintext. This last case is the best, because it enables users to protect their data in any possible situation, but it's also the hardest to design securely: more features often bring more complexity and potential vulnerabilities.

Another piece of functional criteria to consider relates to whether you can use the same core algorithm for both encryption and decryption. For example, many authenticated ciphers are based on the AES block cipher, which specifies the use of two similar algorithms for encrypting and decrypting a block. As discussed in Chapter 4, the CBC block cipher mode requires both algorithms, but the CTR mode requires only the encryption algorithm. Likewise, authenticated ciphers may not need both algorithms. Although the extra cost of implementing both encryption and decryption algorithms won't impact most software, it's often noticeable on low-cost dedicated hardware, where you measure implementation cost in terms of logic gates, or the silicon area occupied by the cryptography.

The AES-GCM Authenticated Cipher Standard

AES-GCM is the most widely used authenticated cipher. AES-GCM is based on the AES algorithm, and the Galois counter mode (GCM) of operation is essentially a tweak of the CTR mode that incorporates a small and efficient component to compute an authentication tag. As I write this, AES-GCM is a NIST standard (SP 800-38D), is part of NSA's Suite B, and is recognized by the IETF for the secure network protocols IPsec, SSH, and TLS 1.2 and 1.3.

NOTE *Although GCM works with any block cipher, you'll probably see it used only with AES.*

GCM Internals

Figure 8-2 shows how AES-GCM works: AES instances parameterized by a secret key (K) transform a block composed of the nonce (N) concatenated with a counter (starting here at 1, then incremented to 2, 3, and so on) and then XOR the result with a plaintext block to obtain a ciphertext block. So far, that's nothing new when compared to the CTR mode.

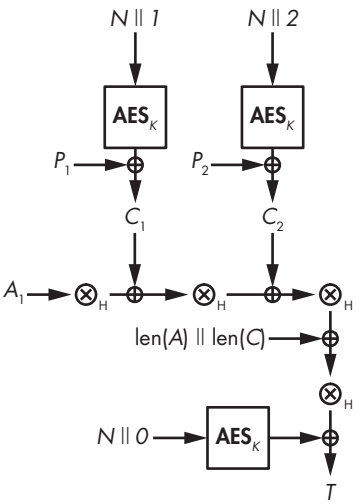


Figure 8-2: The AES-GCM mode, applied to one associated data block, A_1, and two plaintext blocks, P_1 and P_2. The circled multiplication sign represents polynomial multiplication by H, the authentication key derived from K.

The ciphertext blocks are then mixed using a combination of XORs and multiplications (as you'll see next). You can see AES-GCM as doing (1) an encryption in CTR mode and (2) a MAC over the ciphertext blocks. Therefore, AES-GCM is essentially an encrypt-then-MAC construction, where AES-CTR encrypts using a 128-bit key (K) and a 96-bit nonce (N).

To authenticate the ciphertext, GCM uses a Wegman–Carter MAC (see Chapter 7), which XORs the value **AES**(K, N|| 0) with the output of the universal hash function *GHASH*. In Figure 8-2, GHASH corresponds to the series of operations "$\otimes_H$" followed by the XOR with len(A) || len(C), or the bit length of A (the associated data) followed by the bit length of C (the ciphertext).

You can thus express the authentication tag's value as T = **GHASH**(H, A, C) ⊕ **AES**(K, N|| 0), where H is the *hash key*, or *authentication key*. This key is determined as H = **AES**(K, 0), which is the encryption of the block equal to a sequence of null bytes (I didn't include this step in Figure 8-2 for clarity).

NOTE *In GCM, GHASH doesn't use* K *directly in order to ensure that if GHASH's key is compromised, the master key* K *remains secret. Given* K*, you can get* H *by computing* ***AES****(*K*, 0), but you can't recover* K *from that value, since* K *acts here as AES's key.*

GHASH uses *polynomial notation* to multiply each ciphertext block with the authentication key H. This use of polynomial multiplication makes GHASH fast in hardware and software, thanks to a special polynomial multiplication instruction available in many common microprocessors (`CLMUL`, for carry-less multiplication).

Alas, GHASH is far from ideal. For one thing, its speed is suboptimal. Even when using the `CLMUL` instruction, the AES-CTR layer that encrypts the plaintext remains faster than the GHASH MAC. Second, GHASH is painful to implement correctly. In fact, even the experienced developers of the OpenSSL project, by far the most-used cryptographic piece of software in the world, got AES-GCM's GHASH wrong. One commit had a bug in a `gcm_ghash_clmul` function that allowed attackers to forge valid MACs for the AES-GCM. (Fortunately, Intel engineers spotted the error before the bug entered the next OpenSSL release.)

POLYNOMIAL MULTIPLICATION

While more complicated than classic integer arithmetic, polynomial multiplication is simpler for computers because there are no carries. Specifically, you can compute *modular* multiplication of polynomials, in a structure defined by a given polynomial (such structures are called *quotient rings*). For example, say you want to compute the product of the polynomials $(1 + X + X^2)$ and $(1 + X^2)$ modulo $(1 + X + X^3)$. You first multiply the two polynomials $(1 + X + X^2)$ and $(1 + X^2)$ as though you're doing normal polynomial multiplication, thus giving the following (the two terms X^2 cancel each other out because you work with Boolean polynomials, a world wherein 1 + 1 = 0):

$$(1 + X + X^2) \otimes (1 + X^2) = 1 + X + X^3 + X^4$$

(continued)

You apply modulo reduction, reducing $1 + X + X^3 + X^4$ modulo $1 + X + X^3$ to give $X + X^2$, because you can write $1+ X + X^3 + X^4$ as $(1 + X) \otimes (1 + X + X^3) + X + X^2$. In more general terms, $A + BC$ modulo B is equal to A, by definition of modular reduction.

GCM Security

AES-GCM's biggest weakness is its fragility in the face of nonce repetition. If you use the same nonce N twice to encrypt two distinct messages, then an attacker observing the two ciphertexts can determine the value of the XOR between their respective plaintexts. They can also retrieve the authentication key H and use it to forge tags for any ciphertext, associated data, or combination thereof.

A look at the basic algebra behind AES-GCM's computations (see Figure 8-2) helps clarify this fragility. You compute a tag (T) as $T = \mathbf{GHASH}(H, A, C) \oplus \mathbf{AES}(K, N \| 0)$, where GHASH is a universal hash function with linearly related inputs and outputs.

If you compute two tags, T_1 and T_2, with the same nonce N, the AES part will vanish. If you have two tags, $T_1 = \mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{AES}(K, N \| 0)$ and $T_2 = \mathbf{GHASH}(H, A_2, C_2) \oplus \mathbf{AES}(K, N \| 0)$, XORing them together results in the following:

$$\begin{aligned} T_1 + T_2 &= (\mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{AES}(K, N \| 0)) \oplus \\ &\quad (\mathbf{GHASH}(H, A_2, C_2) \oplus \mathbf{AES}(K, N \| 0)) \\ &= \mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{GHASH}(H, A_2, C_2) \end{aligned}$$

If you use the same nonce twice, an attacker can thus recover the value $\mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{GHASH}(H, A_2, C_2)$ for some known A_1, C_1, A_2, and C_2. The linearity of GHASH then allows an attacker to recover H.

In 2016, researchers scanned the internet for instances of AES-GCM exposed through HTTPS servers, in search of systems with repeating nonces (see the research article *https://eprint.iacr.org/2016/475*). They found 184 servers with repeating nonces, including 23 that always used the all-zero string as a nonce.

GCM Efficiency

An advantage of GCM mode is that GCM encryption and decryption process blocks independently, allowing you to parallelize their computation. However, the GMAC computation isn't parallelizable, because it must be computed from the beginning to the end of the ciphertext once GHASH has processed any associated data. This lack of parallelizability means that any system that receives the plaintext first and then the associated data has

to wait until all associated data is read and hashed before hashing the first ciphertext block.

However, GCM is streamable: since the computations in its two layers can be pipelined, there's no need to store all ciphertext blocks before computing GHASH because GHASH processes each block as it's encrypted. In other words, P_1 encrypts to C_1, then GHASH processes C_1 while P_2 encrypts to C_2, then P_1 and C_1 are no longer needed, and so on.

The OCB Authenticated Cipher Mode

First developed in 2001, *offset codebook (OCB)* predates GCM, and like GCM it produces an authenticated cipher from a block cipher, though it does so faster and more simply. OCB has yet to see wider adoption because until 2021, the use of OCB was patented and required a license from its inventor. OCB is now free for anyone to use in any application.

Unlike GCM, OCB blends encryption and authentication into one processing layer that uses a single key. There's no separate authentication layer, so OCB provides authentication mostly for free and performs nearly as many block cipher calls as a nonauthenticated cipher would—OCB is almost as simple as the ECB mode (see Chapter 4), except that it's more secure.

OCB Internals

Figure 8-3 shows how OCB works: it encrypts each plaintext block P to a ciphertext block $C = \mathbf{E}(K, P \oplus O) \oplus O$, where $\mathbf{E}$ is a block cipher encryption function. Here, O (the *offset*) is a value that depends on the key and the nonce incremented for each new block processed.

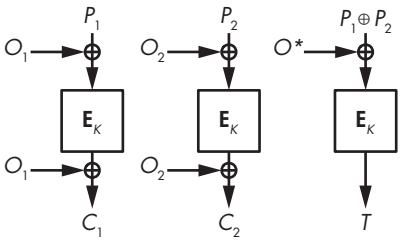


Figure 8-3: The OCB encryption process when run on two plaintext blocks, with no associated data

To produce the authentication tag, OCB first XORs the plaintext blocks together to compute $S = P_1 \oplus P_2 \oplus P_3 \oplus \ldots$ The tag is then $T = \mathbf{E}(K, S \oplus O^*)$, where O^* is an offset value computed from the offset of the last plaintext block processed.

Like AES-GCM, OCB also supports associated data as a series of blocks, A_1, A_2, and so on. When an OCB encrypted message contains associated data, calculate the authentication tag according to the formula

$$T = \mathbf{E}(K, S \oplus O^*) \oplus \mathbf{E}(K, A_1 \oplus O_1) \oplus \mathbf{E}(K, A_2 \oplus O_2) \oplus \ldots$$

where OCB defines different offsets from those used to encrypt *P*.

Unlike GCM and encrypt-then-MAC, which combine ciphertext blocks to form the tag, OCB calculates the authentication tag by combining plaintext data. There's nothing wrong with this approach, and OCB is backed by solid security proofs.

NOTE *For more on how to implement OCB, see either RFC 7253 or the 2011 paper "The Software Performance of Authenticated-Encryption Modes" by Ted Krovetz and Phillip Rogaway, which covers the latest and best version of OCB, OCB3. For further details on OCB, see the FAQ at* http://web.cs.ucdavis.edu/rogaway/ocb/ocb-faq.htm.

OCB Security

OCB is a bit less fragile than GCM against repeated nonces. If you use a nonce twice, an attacker that sees the two ciphertexts could notice that, say, the third plaintext block of the first message is identical to the third plaintext block of the second message. With GCM, attackers can find not only duplicates but also the XOR differences between blocks at the same position. The impact of repeated nonces is therefore worse with GCM than with OCB.

As with GCM, repeated nonces jeopardize the authenticity of OCB, though less potently. For example, an attacker could combine blocks from two messages authenticated with OCB to create another encrypted message with the same checksum and tag as one of the original two messages, but the attacker wouldn't be able to recover a secret key as with GCM.

In 2023, cryptographers discovered that using OCB3 with very short nonces (6 bits) causes its security to significantly reduce; see the article at *https://eprint.iacr.org/2023/326*. Note that the version OCB2 was broken, as described in the article at *https://eprint.iacr.org/2019/311.pdf*.

OCB Efficiency

OCB and GCM are about equally efficient. Like GCM, OCB is parallelizable and streamable. In terms of raw efficiency, GCM and OCB make about as many calls to the underlying block cipher (usually AES), but OCB is slightly faster than GCM because it simply XORs the plaintext rather than performing something like the relatively expensive GHASH computation. (In earlier generations of Intel microprocessors, AES-GCM used to be more than three times slower than AES-OCB because AES and GHASH instructions had to compete for CPU resources and couldn't be run in parallel.)

An important difference between OCB and GCM implementations is that OCB needs both the block cipher's encryption and decryption functions to encrypt and decrypt, increasing the cost of hardware implementations when only limited silicon is available for crypto components. In contrast, GCM uses only the encryption function for both encryption and decryption.

The SIV Authenticated Cipher Mode

Synthetic IV (SIV) is an authenticated cipher mode typically used with AES. Unlike GCM and OCB, SIV is secure even if you use the same nonce twice: an attacker that gets two ciphertexts encrypted using the same nonce would only be able to learn whether the same plaintext was encrypted twice. Unlike with GCM or OCB, the attacker would be unable to tell whether the first block of the two messages is the same, because the nonce used to encrypt is first computed as a combination of the given nonce and the plaintext.

The SIV construction specification is more general than that of GCM. Instead of specifying detailed internals as with GCM's GHASH, SIV simply tells you how to combine a cipher (**E**) and a pseudorandom function (**PRF**) to get an authenticated cipher: you first compute the tag $T = \mathbf{PRF}(K_1, N \| P)$ and then the ciphertext $C = \mathbf{E}(K_2, T, P)$, where T acts as the nonce of **E**. Thus, SIV needs two keys (K_1 and K_2) and a nonce (N).

The major limitation of SIV is that it's not streamable: after computing T, it must keep the entire plaintext P in memory. In other words, to encrypt a 100GB plaintext with SIV, you must first store the 100GB so that SIV encryption can read it.

The document RFC 5297, based on the 2006 paper "Deterministic Authenticated-Encryption" by Phillip Rogaway and Thomas Shrimpton, specifies SIV as using CMAC-AES (a MAC construction using AES) as a PRF and AES-CTR as a cipher. In 2015, a more efficient version of SIV was proposed, GCM-SIV, which combines GCM's fast GHASH function and SIV's mode and is nearly as fast as GCM. Like the original SIV, however, GCM-SIV isn't streamable. (For more information, see *https://eprint.iacr.org/2015/102*.)

Permutation-Based AEAD

Now for a totally different approach to building an authenticated cipher: instead of building a mode of operation around a block cipher like AES, we'll look at a cipher that builds a mode around a permutation. A permutation simply transforms an input to an output of the same size, reversibly, without using a key, that's the simplest component imaginable. Better still, the resulting AEAD is fast, provably secure, and more resistant to nonce reuse than GCM and OCB.

Figure 8-4 shows how a permutation-based AEAD works: from a fixed initial state H_0, you XOR the key K followed by the nonce N to the internal

state, to obtain a new value of the internal state that's the same size as the original. You then transform the new state with a permutation **P** and get a new value of the state. Now you XOR the first plaintext block P_1 to the state and take the resulting value as the first ciphertext block C_1, where P_1 and C_1 are equal in size but shorter than the state.

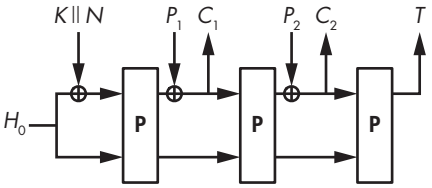


Figure 8-4: A permutation-based authenticated cipher

To encrypt a second block, transform the state with **P**, XOR the next plaintext block P_2 to the current state, and take the resulting value as C_2. Then iterate over all plaintext blocks and, following the last call to **P**, take bits from the internal state as the authentication tag T, as in the right side of Figure 8-4.

NOTE *You can adapt the mode in Figure 8-4 to support associated data, but the process is a bit more complicated, so I'll skip its description.*

Designing *secure* permutation-based authenticated ciphers has certain requirements. First, only XOR input values to a part of the state: the larger this part, the more control a successful attacker has on the internal state and thus the lower the cipher's security. Indeed, all security relies on the secrecy of the internal state.

Also, you must properly pad blocks with extra bits in a way that ensures that any two different messages yield different results. As a counterexample, if the last plaintext block is shorter than a complete block, it shouldn't just be padded with 0s; otherwise, a plaintext block of, say, 2 bytes (0000) would result in a complete plaintext block (0000 . . . 0000), as would a block of 3 bytes (000000). As a result, you'd get the same tag for both messages, although they differ in size.

If you reuse a nonce in such a permutation-based cipher, the impact isn't as bad as with GCM or OCB—the strength of the authentication tag won't be compromised. If you repeat a nonce, an attacker would only learn whether the two encrypted messages begin with the same value, as well as the length of this common value, or prefix. For example, although encrypting the two six-block messages *ABCXYZ* and *ABCDYZ* (each letter symbolizing a block here) with the same nonce might yield the two ciphertexts *JKLTUV* and *JKLMNO*, which have identical prefixes, attackers wouldn't be able to learn that the two plaintexts shared the same final two blocks (*YZ*).

In terms of performance, permutation-based ciphers offer the benefits of a single layer of operations, streamable processing, and the use of a single-core algorithm for encryption and decryption. However, they aren't parallelizable like GCM or OCB because new calls to **P** need to wait for the previous call to complete.

NOTE *If you're tempted to pick your favorite permutation and make up your own authenticated cipher, please don't. You're likely to get the details wrong and end up with an insecure cipher. Read the specifications written by experienced cryptographers for algorithms such as Keyak (an algorithm derived from Keccak), the Duplex construction, and deck functions on* https://keccak.team. *You'll see that permutation-based ciphers are more complex than they first appear.*

How Things Can Go Wrong

Authenticated ciphers have a larger attack surface than hash functions or block ciphers because they aim to achieve both confidentiality *and* authenticity. They take several different input values and must remain secure regardless of whether the input contains only associated data and no encrypted data, extremely large plaintexts, or different key sizes. They must also be secure for all nonce values against attackers who collect numerous message/tag pairs and, to some extent, against accidental repetition of nonces.

That's a lot to ask, and even AES-GCM has several imperfections.

AES-GCM and Weak Hash Keys

One of AES-GCM's weaknesses is in its authentication algorithm GHASH: certain values of the hash key H greatly simplify attacks against GCM's authentication mechanism. Specifically, if the value H belongs to some specific, mathematically defined subgroups of all 128-bit strings, attackers might be able to guess a valid authentication tag for some message simply by shuffling the blocks of a previous message.

To understand this weakness, let's look at how GHASH works.

GHASH Internals

As you saw in Figure 8-2, GHASH starts with a 128-bit value, H, initially set to **AES**(K, 0), and then repeatedly computes

$$X_i = (X_{i-1} \oplus C_i) \otimes H$$

starting from $X_0 = 0$ and processing ciphertext blocks C_1, C_2, and so on. GHASH returns the final X_i to compute the final tag.

Say for the sake of simplicity that all C_i values are equal to 1 so that for any I you have this:

$$C_i \otimes H = 1 \otimes H = H$$

Next, computing X_1 from our first equation yields

$$X_1 = (X_0 \oplus C_1) \otimes H = (0 \oplus 1) \otimes H = H$$

substituting X_0 with 0 and C_1 with 1, to yield the following:

$$(0 \oplus 1) = 1$$

Thanks to the distributive property of $\otimes$ over $\oplus$, we substitute X_1 with H and C_2 with 1 and then compute the next value, X_2, as

$$X_2 = (X_1 \oplus C_2) \otimes H = (H \oplus 1) \otimes H = H^2 \oplus H$$

where H^2 is H squared, or $H \otimes H$.

Now you derive X_3 by substituting X_2 for its derivation, and obtain the following:

$$X_3 = \left(X_2 \oplus C_3\right) \otimes H = \left(H^2 \oplus H \oplus 1\right) \otimes H = H^3 \oplus H^2 \oplus H$$

Next, you derive X_4 to be $X_4 = H^4 \oplus H^3 \oplus H^2 \oplus H$, and so on, and eventually the last X_i is this:

$$X_n = H^n \oplus H^{n-1} \oplus H^{n-2} \oplus \ldots \oplus H^2 \oplus H$$

Remember that you set all blocks C_i equal to 1. If these values are arbitrary instead, you end up with the following:

$$X_n = (C_1 \otimes H^n) \oplus (C_2 \otimes H^{n-1}) \oplus (C_3 \otimes H^{n-2}) \oplus \ldots \oplus (C_{n-1} \otimes H^2) \oplus (C_n \otimes H)$$

GHASH then XORs the message's length to this last X_n, multiplies the result by H, and XORs this value with $\mathbf{AES}(K, N \| 0)$ to create the final authentication tag, T.

Where Things Break

What can go wrong from here? Let's look first at the simplest cases:

- If $H = 0$, then $X_n = 0$ regardless of the C_i values and thus regardless of the message. That is, all messages will have the same authentication tag if H is 0.
- If $H = 1$, then the tag is just an XOR of the ciphertext blocks, and reordering the ciphertext blocks will give the same authentication tag.

Since 0 and 1 are only two of 2^{128} possible values of H, there's only a $2/2^{128} = 1/2^{127}$ chance of these occurring. But there are other weak values as well—namely, all values of H that belong to a *short cycle* when raised to ith powers. For example, the value of H `10d04d25f93556e69f58ce2f8d035a4` belongs to a cycle of length 5, as it satisfies $H^5 = H$, and therefore $H^e = H$ for any e that is a multiple of 5 (the very definition of cycle with respect to fifth powers). Consequently, in the preceding expression of the final GHASH

value X_n, swapping the blocks C_n (multiplied to H) and the block C_{n-4} (multiplied to H^5) leaves the authentication tag unchanged, which amounts to a forgery. An attacker may exploit this property to construct a new message and its valid tag without knowing the key, which should be impossible for a secure authenticated cipher.

The preceding example is based on a cycle of length 5, but there are many cycles of greater length and therefore many values of H that are weaker than they should be. The upshot is that, in the unlikely case that H belongs to a short cycle of values and attackers can forge as many authentication tags as they want, unless they know H or K, they can't determine H's cycle length. While attackers can't exploit this vulnerability, you can avoid it by carefully choosing the polynomial used for modulo reductions.

NOTE *For further details on this attack, read "Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes" by Markku-Juhani O. Saarinen, available at* https://eprint.iacr.org/2011/202.

AES-GCM and Small Tags

In practice, AES-GCM usually returns 128-bit tags, but it can produce tags of any length. Unfortunately, when using shorter tags, the probability of forgery increases significantly.

When using a 128-bit tag, an attacker who attempts a forgery should succeed with a probability of $1/2^{128}$ because there are 2^{128} possible 128-bit tags. (Generally, with an n-bit tag, the probability of success should be $1/2^n$, where 2^n is the number of possible values of an n-bit tag.) But when using shorter tags, the probability of forgery is much higher than $1/2^n$ due to weaknesses in the structure of GCM that are beyond the scope of this discussion. For example, a 32-bit tag allows an attacker who knows the authentication tag of some 2MB message to succeed with a chance of $1/2^{16}$ instead of $1/2^{32}$.

Generally, with n-bit tags, the probability of forgery isn't $1/2^n$ but rather $2^m/2^n$, where 2^m is the number of blocks of the longest message for which a successful attacker observed the tag. For example, if you use 48-bit tags and process messages of 4GB (or 2^{28} blocks of 16 bytes each), the probability of a forgery is $2^{28}/2^{48} = 1/2^{20}$, or about one chance in a million. That's a relatively high chance as far as cryptography is concerned. (For more information on this attack, see the 2005 paper "Authentication Weaknesses in GCM" by Niels Ferguson.)

Further Reading

To learn more about authenticated ciphers, visit the home page of CAESAR, the Competition for Authenticated Encryption: Security, Applicability, and Robustness (*http://competitions.cr.yp.to/caesar.html*). Started in 2012, CAESAR was a crypto competition in the style of the AES and SHA-3 competitions, though it isn't organized by NIST.

The CAESAR competition attracted an impressive number of innovative designs: from OCB-like modes to permutation-based modes (such as NORX and Keyak), as well as totally original algorithms (such as AEZ or AEGIS). In 2019, CAESAR concluded with the selection of a portfolio of seven algorithms divided into three categories, which you can find on its official page at *https://competitions.cr.yp.to/caesar-submissions.html*.

Another competition where you can find authenticated encryption algorithms was NIST's Lightweight Cryptography project. Running from 2017 to 2023, it aimed at standardizing algorithms optimized for resource-constrained environments (memory, processor size, and so on) such as embedded platforms. The winner of the competition was Ascon, a family of permutation-based algorithms, presented in detail on its official site at *https://ascon.iaik.tugraz.at*. To discover the other candidate algorithms and the work presented at the project workshops, visit *https://csrc.nist.gov/Projects/lightweight-cryptography*.

In this chapter, we focused on GCM, but a handful of other modes are used in real applications as well. Specifically, the counter with CBC-MAC (CCM) and EAX modes competed with GCM for standardization in the early 2000s, and although GCM was selected, its competitors are used in a few applications. For example, CCM is used in the WPA2 Wi-Fi encryption protocol. Consider reading these ciphers' specifications and reviewing their relative security and performance merits.

This concludes our discussion of symmetric-key cryptography! You've seen block ciphers, stream ciphers, (keyed) hash functions, and now authenticated ciphers—all the main cryptography components that work with a symmetric key, or no key at all. Before we move to *asymmetric* cryptography, Chapter 9 focuses on computer science and math, to provide background for asymmetric schemes such as RSA (Chapter 10) and Diffie–Hellman (Chapter 11).

PART III

ASYMMETRIC CRYPTO

9

HARD PROBLEMS

Hard computational problems are the cornerstone of modern cryptography. These are problems for which even the best algorithm wouldn't find a solution before the sun burns out.

In the 1970s, the rigorous study of hard problems gave rise to a new field of science called *computational complexity theory*, which dramatically impacted cryptography and many other fields, including economics, physics, and biology. In this chapter, you'll learn the conceptual tools from complexity theory necessary to understand the foundations of cryptographic security. I'll also introduce the hard problems behind public-key schemes, such as RSA encryption and Diffie–Hellman key agreement. I'll touch on some deep concepts, but I'll minimize the technical details and scratch only the surface. Still, I hope you'll see the beauty in how cryptography leverages computational complexity theory to maximize security assurance.

Computational Hardness

A computational problem is a question that one can answer by doing enough computation—for example, "Is 217 a prime number?" or "How many *i*s are in *incomprehensibilities*?" The first question is a decision problem, because it can be answered with "yes" or "no," while the second is a search problem.

Computational hardness is the property of computational problems for which there is no algorithm that will run in a reasonable amount of time. Such problems are also called *intractable*. Computational hardness is independent of the type of computing device used, be it a general-purpose central processing unit (CPU), a graphics processing unit (GPU), an integrated circuit, or a mechanical Turing machine. Indeed, one of the first findings of computational complexity theory is that all computing models are equivalent. If one computing device can solve a problem efficiently, any other device can efficiently solve it by porting the algorithm to the other device's language—an exception is quantum computers, which we'll discuss in Chapter 14. Thus, I won't need to specify the underlying computing device or hardware when discussing computational hardness; instead, we'll just discuss algorithms.

To evaluate computational hardness, you first need a way to measure the complexity of an algorithm, or its running time. You then categorize running times as hard or easy.

Running Time

The *computational complexity* of an algorithm is the approximate number of operations it does, as a function of its input size. You can count the size in bits or in the number of elements taken as input. For example, take the algorithm in Listing 9-1, written in pseudocode. It searches for a value, *x*, within an array of *n* elements and then returns its index position, or –1, if *x* isn't found.

```
search(x, array, n) {
    for i from 0 to n - 1 {
        if (array[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Listing 9-1: A simple search algorithm of complexity linear with respect to the array length `n`

This algorithm uses a `for` loop to find a specific value, *x*, in an array, iterating over values of the variable *i*, starting with 0. It checks whether the value of position *i* in `array` is equal to the value of *x*. If so, it returns the position *i*. Otherwise, it increments *i* and tries the next position until it reaches $n - 1$, the last position in the array, at which point it returns –1.

For this kind of algorithm, you count complexity as the number of iterations of the `for` loop: 1 in the best case (if x is equal to `array[0]`), n in the worst case (if x is equal to `array[n - 1]` or if x isn't found in `array`), and $n/2$ on average if x is uniformly randomly distributed in one of the n cells of the array. With an array 10 times as large, the algorithm will be 10 times as slow. Complexity is therefore proportional to n, or "linear" in n. A complexity linear with respect to its input size is considered fast, as opposed to exponential complexities. In this example, although processing larger input values is slower, the computational cost won't blow up exponentially but remains proportional to the table size.

However, many useful algorithms are slower than that and have a complexity higher than linear. The textbook example is sorting algorithms: given a list of n values in a random order, you need in the worst case $n \times \log n$ basic operations to sort the list, which is sometimes called *linearithmic complexity*. Since $n \times \log n$ grows faster than n, sorting speed slows down faster than proportionally to n. Yet such sorting algorithms remain in the realm of *practical* computation, or computation that one can carry out in a reasonable amount of time.

What's usually *not* reasonable are complexities exponential in the input size. At some point, you'll hit the ceiling of what's feasible even for relatively small input lengths. Take the simplest example from cryptanalysis: the brute-force search for a secret key. Recall from Chapter 1 that given a plaintext P and a ciphertext $C = \mathbf{E}(K, P)$, it takes at most 2^n attempts to recover an n-bit symmetric key because there are 2^n possible keys—an example of a complexity that grows exponentially. A problem with *exponential complexity* is practically impossible to solve because as n grows, the effort rapidly becomes infeasible.

You may object that we're comparing oranges and apples here: in the `search()` function in Listing 9-1, we counted the number of `if (array[i] == x)` operations, whereas key recovery counts the number of encryptions, each thousands of times slower than a single `==` comparison. This seeming inconsistency can make a difference if you compare two algorithms with very similar complexities, but most of the time it won't matter because the number of operations has a greater impact than the cost of an individual operation. Also, complexity estimates ignore *constant factors*: when we say that an algorithm takes time in the order of n^3 operations (which is *cubic complexity*), it may actually take $41 \times n^3$ operations, or even $12{,}345 \times n^3$ operations—but again, as n grows, the constant factors lose significance to the point that you can ignore them. Complexity analysis is about theoretical hardness as a function of the input size; it doesn't care about the exact number of CPU cycles it takes on your computer.

You can often use the $O()$ notation (*big O*) to express complexities. For example, $O(n^3)$ means that complexity grows no faster than n^3, ignoring potential constant factors. $O()$ denotes the *upper bound* of an algorithm's complexity. The notation $O(1)$ means that an algorithm runs in *constant time*—that is, the running time doesn't depend on the input length. For example, the algorithm that determines an integer's parity by looking at its least significant bit (LSB) and returning "even" if it's zero and "odd"

otherwise will do the same thing at the same cost, whatever the integer's length.

To see the difference between linear, quadratic, and exponential time complexities, look at how complexity grows for $O(n)$ (linear) versus $O(n^2)$ (quadratic) versus $O(2^n)$ (exponential) in Figure 9-1.

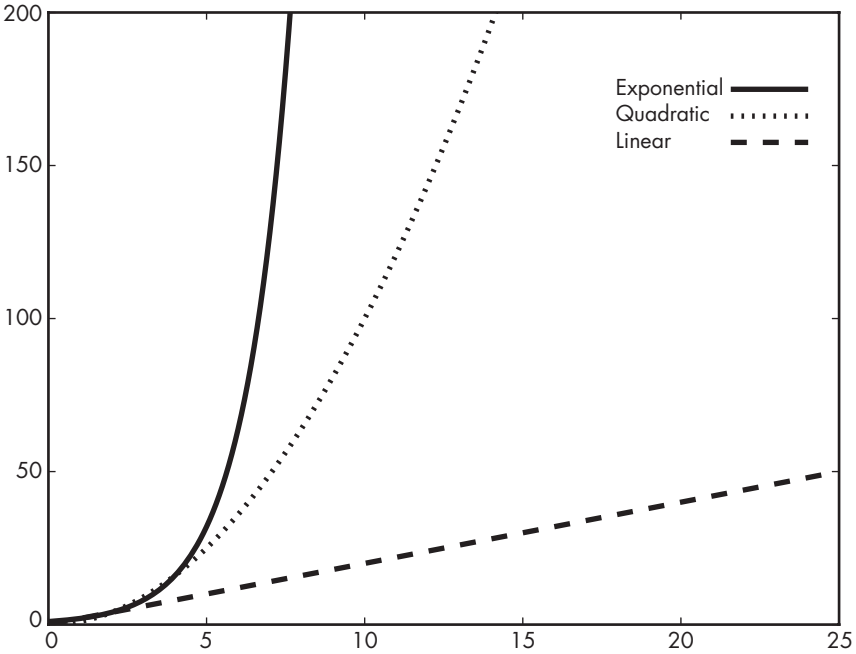


Figure 9-1: The growth of exponential, quadratic, and linear complexities, from the fastest to the slowest growing

Exponential complexity means the problem is practically impossible to solve, and linear complexity means the solution is feasible, whereas quadratic complexity is somewhere between the two.

Polynomial vs. Superpolynomial Time

The quadratic $O(n^2)$ complexity (the middle curve in Figure 9-1) is a special case of the broader class of polynomial complexities, or $O(n^k)$, where k is some fixed number such as 3, 2.373, 7/10, or the square root of 17. Polynomial-time algorithms are eminently important in complexity theory and in cryptography because they're the very definition of practically feasible. When an algorithm runs in *polynomial time*, or *polytime* for short, it completes in a decent amount of time even if the input is large. That's why polynomial time is synonymous with "efficient" for complexity theorists and cryptographers.

In contrast, you can view algorithms running in *superpolynomial time*—that is, in $O(f(n))$, where $f(n)$ is any function that grows faster than any polynomial—as impractical. I'm saying superpolynomial, and not just

exponential, because there are complexities in between polynomial and the well-known exponential complexity $O(2^n)$, such as $O(n^{\log(n)})$, as Figure 9-2 shows.

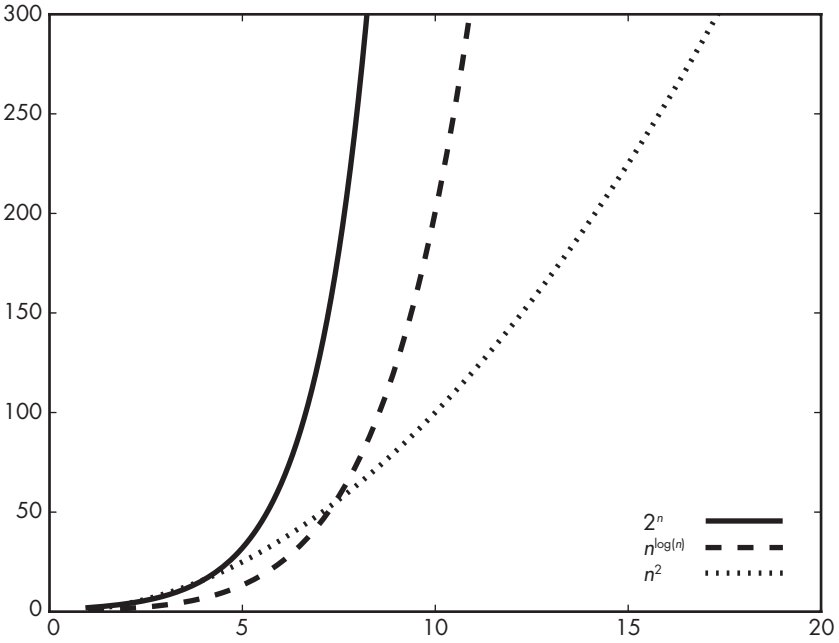


Figure 9-2: The growth of the 2^n, $n^{\log(n)}$, and n^2 functions, from the fastest to the slowest growing

$O(n^2)$ or $O(n^3)$ may be efficient, but $O(n^{99,999,999,999})$ obviously isn't. In other words, polytime is fast in practice as long as the exponent isn't too large. Fortunately, all polynomial-time algorithms found to solve actual problems have small exponents. For example, $O(n^{2.373})$ is the time complexity of the best known algorithm for multiplying two $n \times n$ matrices in theory, as this algorithm is never used in practice. The 2002 breakthrough polytime deterministic algorithm for identifying n-bit prime numbers initially had a complexity $O(n^{12})$, but it was later improved to $O(n^6)$. Polynomial time thus may not be the perfect definition of a practical time for an algorithm, but it's the best we have.

By extension, you can consider a problem that can't be solved by a polynomial-time algorithm impractical, or *hard*. As an example, for a straightforward key search, there's no way to beat the $O(2^n)$ complexity unless the cipher is somehow broken.

NOTE *Exponential complexity* $O(2^n)$ *is not the worst you can get. Some complexities grow even faster and thus characterize algorithms even slower to compute—for example, the complexity* $O(n^n)$ *or the* exponential factorial $O(n^{f(n-1)})$*, where for any* x*, the function* f *is here recursively defined as* $f(x) = x^{f(x-1)}$*. In practice, you'll never encounter algorithms with such preposterous complexities.*

You know that there's no way to beat the $O(2^n)$ complexity of a brute-force key search (as long as the cipher is secure), but you won't always know the fastest way to solve a computational problem in general. A large portion of the research in complexity theory is about proving complexity *bounds* on the running time of algorithms solving a given problem. To make their job easier, complexity theorists have categorized computational problems in different groups, or *classes*, according to the effort needed to solve them.

Complexity Classes

In mathematics, a *class* is a group of objects with some similar attribute. For example, all computational problems solvable in time $O(n^2)$, which complexity theorists simply denote **TIME**(n^2), are one class. Likewise, **TIME**(n^3) is the class of problems solvable in time $O(n^3)$, **TIME**(2^n) is the class of problems solvable in time $O(2^n)$, and so on. For the same reason that a supercomputer can compute whatever a laptop can compute, any problem solvable in $O(n^2)$ is also solvable in $O(n^3)$. Hence, any problem in the class **TIME**(n^2) also belongs to the class **TIME**(n^3), which also both belong to the class **TIME**(n^4), and so on. The union of all the classes **TIME**(n^k), for all constants k, is **P**, which stands for polynomial time.

If you've programmed a computer, you'll know that seemingly fast algorithms may still crash your system by eating all its memory resources. When selecting an algorithm, you should consider not only its time complexity but also how much memory it uses, or its *space complexity*. This is especially important because a single memory access is usually orders of magnitudes slower than a basic arithmetic operation in a CPU.

Formally, you define an algorithm's memory consumption as a function of its input length, n, in the same way you defined time complexity. The class of problems solvable using $f(n)$ bits of memory is **SPACE**$(f(n))$. For example, **SPACE**(n^3) is the class of problems solvable using of the order of n^3 bits of memory. Just as you had **P** as the union of all **TIME**(n^k), the union of all **SPACE**(n^k) problems is **PSPACE**.

While the lower the memory the better, a polynomial amount of memory doesn't necessarily imply that an algorithm is practical. Take, for example, a brute-force key search, which takes negligible memory but is slow as hell. More generally, an algorithm can take forever, even if it uses just a few bytes of memory.

Any problem solvable in time $f(n)$ needs at most $f(n)$ memory, so **TIME**$(f(n))$ is included in **SPACE**$(f(n))$. In time $f(n)$, you can write up to $f(n)$ bits, and no more, because writing (or reading) 1 bit is assumed to take one unit of time; therefore, any problem in **TIME**$(f(n))$ can't use more than $f(n)$ space. As a consequence, **P** is a subset of **PSPACE**.

Nondeterministic Polynomial Time

NP, *nondeterministic* polynomial time, is the second most important complexity class, after the class **P** of all polynomial-time algorithms.

NP is the class of decision problems for which you can verify a solution in polynomial time—that is, efficiently—even though the solution may be hard to find. By *verified*, I mean that given a potential solution, you can run some polynomial-time algorithm that checks whether you've found an actual solution. For example, the problem of deciding whether there exists a key K such that $C = \mathbf{E}(K, P)$ given P and C for a symmetric cryptosystem **E** is in **NP**. This is because given a candidate key K_0, you can check that K_0 is the correct key by verifying that $\mathbf{E}(K_0, P)$ equals C. You can't find a potential key (the solution), if it exists, in polynomial time, but you can check whether a key is correct.

Now for a counterexample: What about known-ciphertext attacks? This time, you get only some $\mathbf{E}(K, P)$ values for random unknown plaintext Ps. If you don't know what the Ps are, then there's no way to verify whether a potential key, K_0, is the right one. In other words, the key-recovery problem under known-ciphertext attacks is not in **NP** (let alone in **P**), as you can't express it as a decision problem.

Another example of a problem not in **NP** is that of verifying the *absence* of a solution to a problem. Verifying that a solution is correct boils down to computing some algorithm with the candidate solution as an input and then checking the return value. However, to verify that *no* solution exists, you may need to go through all possible inputs. If there's an exponential number of inputs, you won't be able to efficiently prove that no solution exists. The absence of a solution is hard to show for the hardest problems in the class **NP**—the so-called **NP**-complete problems.

NP-Complete Problems

***NP**-complete* problems are the hardest decision problems in the class **NP**; you won't know how to solve the worst-case instances of these problems in polynomial time. As complexity theorists discovered in the 1970s when they developed the theory of **NP**-completeness, **NP**'s hardest problems are all fundamentally equally hard. This was proven by showing that you can turn any efficient solution to any of the **NP**-complete problems into an efficient solution for any of the other **NP**-complete problems. In other words, if you can solve any **NP**-complete problem efficiently, you can solve all of them, as well as all problems in **NP**. How can this be?

NP-complete problems come in different guises, but they're fundamentally similar from a mathematical perspective. In fact, you can reduce any **NP**-complete problem to any other **NP**-complete problem such that the capability to solve the second implies the capability to solve the first. Remember that **NP** contains decision problems, not search problems. You can efficiently transform an algorithm able to solve a search problem into an algorithm able to solve the corresponding decision problem, though the converse direction is not always possible. Fortunately, this is the case for NP-complete problems, which explains why people often mix up the two.

Here are some examples of **NP**-complete problems:

The traveling salesman problem Given a set of points on a map (such as cities) with the distances between each point from each other point and given a maximum distance *x*, decide whether there is a path that visits every point such that the total distance is smaller than *x*. (Note that you can find such a path with essentially the same complexity as the decision problem, but deciding whether a path is optimal is not in **NP**, because you can't efficiently verify a solution's correctness.)

The clique problem Given a number, *x*, and a graph (a set of nodes connected by edges, as in Figure 9-3), determine whether there's a set of at most *x* nodes that are all connected to each other.

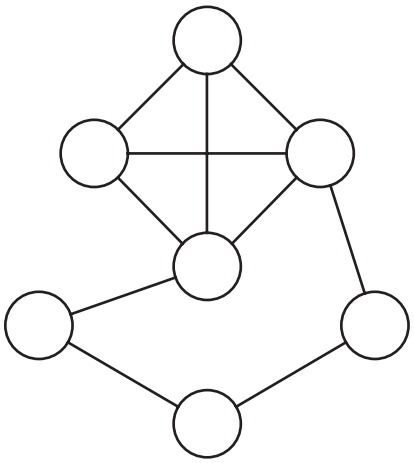


*Figure 9-3: A graph containing a clique of four nodes. The general problem of finding a clique (set of nodes all connected to each other) of a given size in a graph is **NP**-complete.*

The knapsack problem Given two numbers, *x* and *y*, and a set of items, each of a known value and weight, decide if there is a group of items such that the total value is at least *x* and the total weight is at most *y*.

You can find such **NP**-complete problems everywhere, from scheduling (given jobs of some priority and duration and one or more processors, assign jobs to the processors by respecting the priority while minimizing total execution time) to constraint-satisfaction (determine values that satisfy a set of mathematical constraints, such as logical equations). Even the task of winning certain video games was proven to be **NP**-hard (for famous games including *Tetris*, *Super Mario Bros.*, *Pokémon*, and *Candy Crush Saga*). For example, the article "Classic Nintendo Games are (Computationally)

Hard" considers "the decision problem of reachability" to determine the possibility of reaching the goal point from a particular starting point (*https://arxiv.org/abs/1203.1895*).

Some of these video game problems are at least as hard as **NP**-complete problems and are called **NP**-*hard*. A (not necessarily decisional) problem is **NP**-hard when it's at least as hard as **NP**-complete (decision) problems and if any method to solve it can be efficiently used to solve **NP**-complete problems.

NP-complete problems must be decisional problems; that is, problems with a yes or no answer. Therefore, strictly speaking, problems of computing the "best" value of a solution cannot be **NP**-complete but may be **NP**-hard. For example, take the traveling salesman problem: the problem "Is there a path visiting all points with a distance less than *X*?" is **NP**-complete, whereas "Find the faster path visiting all points" is **NP**-hard.

Note that not all *instances* of **NP**-hard problems are actually hard to solve. You may be able to efficiently solve some instances because they're small or have a specific structure. Take, for example, the graph in Figure 9-3. You can quickly spot the clique, which is the top four connected nodes—even though the aforementioned clique-finding problem is **NP**-hard, there's nothing hard here. Being **NP**-hard doesn't mean that all instances of a given problem are hard but that as the problem size grows, some of them are. This is why cryptographers are more interested in problems that are hard on average, not just in the worst case.

The P vs. NP Problem

If you could solve the hardest **NP** problems in polynomial time, then you could solve *all* **NP** problems in polynomial time, and therefore **NP** would equal **P**. Such an equality sounds preposterous: Aren't there problems for which a solution is easy to verify but hard to find? For example, isn't it obvious that exponential-time brute force is the fastest way to recover the key of a symmetric cipher, and therefore that the problem can't be in **P**?

As crazy as it sounds, no one has proved that **P** is different from **NP**, despite a bounty of $1 million. The Clay Mathematics Institute will award this to anyone who proves that either **P** ≠ **NP** or **P** = **NP**. This problem, known as **P** vs. **NP**, was called "one of the deepest questions that human beings have ever asked" by renowned complexity theorist Scott Aaronson. Think about it: if **P** were equal to **NP**, then any easily checked solution would also be easy to find. All cryptography used in practice would be insecure because you could recover symmetric keys and invert hash functions efficiently.

But don't panic: most complexity theorists believe **P** isn't equal to **NP** and therefore that **P** is instead a strict subset of **NP**, as Figure 9-4 shows, where **NP**-complete problems are another subset of **NP** not overlapping with **P**. In other words, problems that look hard actually are hard. It's just difficult to prove this mathematically. While proving that **P** = **NP** requires only a polynomial-time algorithm for an **NP**-complete problem, proving the nonexistence of such an algorithm is fundamentally harder. This didn't

stop mathematicians from coming up with simple proofs that, while usually obviously wrong, often make for funny reads; for an example, see "The P-versus-NP page" (*https://www.win.tue.nl/~wscor/woeginger/P-versus-NP.htm*).

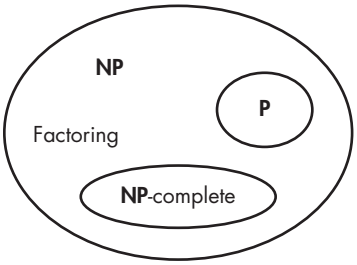


*Figure 9-4: The classes **NP** and **P** and the set of **NP**-complete problems*

If we're almost sure that hard problems exist in **NP**, what about leveraging them to build strong, provably secure crypto? Imagine a proof that breaking some cipher is **NP**-hard and therefore that the cipher is unbreakable as long as **P** isn't equal to **NP**. But reality is disappointing: the search versions of **NP**-complete problems have proved difficult to use for crypto purposes because the very structure that makes them hard in the worst case can make them easy in specific cases that sometimes occur in crypto. Instead, cryptography often relies on problems that are *probably not* **NP**-hard.

The Factoring Problem

The factoring problem consists of finding the prime numbers p and q given a large number, $N = p \times q$. The widely used RSA algorithms are based on the difficulty of factoring: RSA encryption and signature schemes are secure because factoring is a hard problem. Before we see how RSA leverages the factoring problem in Chapter 10, I'd like to convince you that the decision version of this problem ("Does N have a factor smaller than k that is not equal to 1?") is indeed hard yet probably not **NP**-complete.

First, some basic math. A *prime number* isn't divisible by any other number but itself and 1. For example, the numbers 3, 7, and 11 are prime; the numbers 4 (that is, 2×2), 6 (2×3), and 12 ($2 \times 2 \times 3$) are not. A fundamental theorem of number theory says that you can write any integer number uniquely as a product of primes, a representation called the *factorization* of that number. For example, the factorization of 123,456 is $2^6 \times 3 \times 643$, the factorization of 1,234,567 is $127 \times 9{,}721$, and so on. Any integer has a unique factorization, or a unique way to write it as a product of prime numbers. Polynomial-time primality testing algorithms allow us to efficiently test whether a given number is prime or a given factorization contains only prime numbers. Getting from a number to its prime factors, however, is another matter.

Factoring Large Numbers

So how do you go from a number to its factorization—namely, its decomposition as a product of prime numbers? The most basic way to factor a number, N, is to try dividing it by all the numbers lower than it until you find a number, x, that divides N. Then attempt to divide N with the next number, $x + 1$, and so on. You'll end up with a list of factors of N. What's the time complexity of this? First, remember that we express complexities as a function of the input's *length*. The bit length of the number N is $n = \log_2 N$. By definition of the logarithm, this means that $N = 2^n$. Because all the numbers less than $N/2$ are reasonable guesses for possible factors of N, there are about $N/2 = 2^n/2$ values to try. The complexity of our naive factoring algorithm is therefore $O(2^n)$, ignoring the $1/2$ coefficient in the $O()$ notation.

While many numbers are easy to factor by first finding any small factors (2, 3, 5, and so on) and then iteratively factoring any other nonprime factors, here we're interested in numbers of the form $N = p \times q$, where p and q are large, as found in cryptography.

Let's be a bit smarter: as we don't need to test all numbers lower than $N/2$, but rather only the prime numbers, we can start by trying those smaller than the square root of N. If N isn't a prime number, then it must have at least one factor lower than its square root $\sqrt{N}$. This is because if both of N's factors p and q were greater than $\sqrt{N}$, then their product would be greater than $\sqrt{N} \times \sqrt{N} = N$, which is impossible. For example, if $N = 100$, its factors p and q can't both be greater than 10 because that would result in a product greater than 100. Either p or q has to be less than or equal to $\sqrt{N}$.

So what's the complexity of testing only the primes less than $\sqrt{N}$? The *prime number theorem* states that there are approximately $N/\log N$ primes smaller than N. Hence, there are approximately $\sqrt{N}/\log \sqrt{N}$ primes smaller than $\sqrt{N}$. Expressing this value in terms of $n = \log_2 N$, we get approximately $2^{n/2+1}/n$ possible prime factors and therefore a complexity of $O(2^{n/2}/n)$, since $\sqrt{N} = 2^{n/2}$ and $1/\log \sqrt{N} = 1/(n/2) = 2/n$. This is faster than testing all prime numbers, but it's still painfully slow—on the order of 2^{120} operations for a 256-bit number. That's quite an impractical computational effort. But we can do much better.

The fastest factoring algorithm is the *general number field sieve (GNFS)*, which I won't describe here because it requires the introduction of several advanced mathematical concepts. A rough estimate of GNFS's complexity is $\exp(1.91 \times n^{1/3} (\log n)^{2/3})$, where $n = \log_2 N$ is the bit length of N and $\exp(\ldots)$ is just a different notation for the exponential function e^x, with e the exponential constant approximately equal to 2.718. However, it's difficult to get an accurate estimate of GNFS's actual complexity for a given number size. Therefore, we must rely on heuristic complexity estimates, which show how security increases with a longer n. For example:

- Factoring a **1,024-bit** number, which has two prime factors of approximately 500 bits each, takes on the order of 2^{70} basic operations.
- Factoring a **2,048-bit** number, which has two prime factors of approximately 1,000 bits each, takes on the order of 2^{90} basic operations—about a million times slower than for a 1,024-bit number.

You can estimate that reaching 128-bit security requires at least 4,096 bits. Take these values with a grain of salt, as researchers don't always agree on these estimates. The following experimental results reveal the actual cost of factoring:

- In 2005, after about 18 months of computation—and thanks to the power of a cluster of 80 processors, with a total effort equivalent to 75 years of computation on a single processor—a group of researchers factored a **663-bit** (200-decimal digit) number.
- In 2009, after about two years and using several hundred processors, with a total effort equivalent to about 2,000 years of computation on a single processor, another group of researchers factored a **768-bit** (232-decimal digit) number.
- In 2020, after a few months of computation, using tens of thousands of processors and a supercomputer, for a total effort equivalent to around 2,700 years of calculation on a single processor, another team factored an **829-bit** (250-decimal digit) number.

As you can see, the numbers factored by researchers are shorter than those in real applications, which are at least 1,024-bit and often more than 2,048-bit. As I write this, no one has reported the factoring of a 1,024-bit number, but many speculate that well-funded organizations such as the NSA can do it.

In sum, you should view 1,024-bit RSA as insecure and use RSA with at least a 2,048-bit value, but preferably a 4,096-bit one to ensure higher security.

Factoring Is Probably Not NP-Hard

We don't know how to factor large numbers efficiently, which suggests that the factoring problem doesn't belong to **P**. However, the decision version of factoring is clearly in **NP**, because given a factorization, we can verify the solution by checking that all factors are prime numbers, thanks to the aforementioned primality testing algorithm, and that when multiplied together, the factors give the expected number. For example, to check that 3×5 is the factorization of 15, confirm that both 3 and 5 are prime and that 3 times 5 equals 15.

So we have a problem that is in **NP** and that looks hard, but is it as hard as the hardest **NP** problems? In other words, is the decision version of factoring **NP**-complete, and as a consequence, is factoring **NP**-hard? Spoiler alert: probably not.

There's no mathematical proof that factoring isn't **NP**-hard, but we have a few pieces of soft evidence. First, all known **NP**-hard problems in **NP** can have one solution, more than one solution, or no solution at all. In contrast, factoring always has exactly one solution. Also, the factoring problem has a mathematical structure that allows for the GNFS algorithm to significantly outperform a naive algorithm, a structure that **NP**-hard problems don't have. Factoring would be easy with a *quantum computer*, a computing model that exploits quantum mechanical phenomena to run different kinds of

algorithms and that would have the capability to factor large numbers efficiently (not because it'd run the algorithm faster but because it could run a quantum algorithm dedicated to factoring large numbers). Such a quantum computer doesn't exist yet, though—and might never. Regardless, a quantum computer is believed to be useless in tackling **NP**-hard problems because it'd be no faster than a classical one (see Chapter 14).

Factoring may be slightly easier than solving **NP**-hard problems in theory, but as far as cryptography is concerned, it's hard enough, and even more reliable than **NP**-hard problems. Indeed, it's easier to build cryptosystems on top of the factoring problem than search versions of **NP**-complete problems because it's difficult to know exactly how hard it is to break a cryptosystem based on such problems—in other words, how many bits of security you'd get. As mentioned earlier, this relates to the fact that **NP** concerns worst-case hardness, and cryptographers are looking for average-case hardness.

The factoring problem is one of several problems you can use in cryptography as a *hardness assumption*, which is an assumption that some problem is computationally hard. You can use this assumption when proving that breaking a cryptosystem's security is at least as hard as solving said problem. Another problem you can use as a hardness assumption, the *discrete logarithm problem (DLP)*, is actually a family of problems.

The Discrete Logarithm Problem

The Discrete Logarithm Problem (DLP) predates the factoring problem in the official history of cryptography. Whereas RSA appeared in 1977, another cryptographic breakthrough, the Diffie–Hellman key agreement (see Chapter 11), came about a year earlier, grounding its security on the hardness of the DLP. Like the factoring problem, the DLP deals with large numbers, but it's less straightforward and requires more math than factoring. Let's begin by introducing the mathematical notion of a group in the context of discrete logarithms.

Groups

In a mathematical context, a *group* is a set of elements (typically, numbers) that relate to each other according to certain well-defined rules. An example of a group is the set of nonzero integers modulo a prime number p (that is, numbers between 1 and $p - 1$) with modular multiplication being the group operation. We note such a group as $(\mathbf{Z}_p^*, \times)$, or just $\mathbf{Z}_p^*$ when it's clear that the group operation is multiplication.

For $p = 5$, you get the group $\mathbf{Z}_5^* = \{1, 2, 3, 4\}$. In the group $\mathbf{Z}_5^*$, operations are carried out modulo 5; hence, you don't have $3 \times 4 = 12$ but instead $3 \times 4 = 2$, because 12 mod 5 = 2. You can nonetheless use the same sign ($\times$) that you use for normal integer multiplication. You can also use the exponent notation to denote a group element's multiplication with itself mod p, a common operation in cryptography. For example, in the context of $\mathbf{Z}_5^*$, $2^3 = 2 \times 2 \times 2 = 3$ rather than 8, because 8 mod 5 is equal to 3.

To be a group, a mathematical set and its operation must have the following characteristics, which are called *group axioms*:

Closure For any two group elements x and y, $x \times y$ is in the group too. In $\mathbf{Z}_5^*$, $2 \times 3 = 1$ (because $6 = 1 \bmod 5$), $2 \times 4 = 3$, and so on.

Associativity For any group elements x, y, z, $(x \times y) \times z = x \times (y \times z)$. In $\mathbf{Z}_5^*$, $(2 \times 3) \times 4 = 1 \times 4 = 2 \times (3 \times 4) = 2 \times 2 = 4$.

Identity existence The group includes an element e such that $e \times x = x \times e = x$. Such an element is called the *identity*. In any $\mathbf{Z}_p^*$, the identity element is 1.

Inverse existence For any x in the group, there's a y such that $x \times y = y \times x = e$. In $\mathbf{Z}_5^*$, the inverse of 2 is 3, and the inverse of 3 is 2, while 4 is its own inverse because $4 \times 4 = 16 = 1 \bmod 5$.

In addition, a group is *commutative*, or *abelian*, if $x \times y = y \times x$ for any group elements x and y. That's also true for any multiplicative group of integers $\mathbf{Z}_p^*$. In particular, $\mathbf{Z}_5^*$ is commutative: $3 \times 4 = 4 \times 3$, $2 \times 3 = 3 \times 2$, and so on.

A group is *cyclic* if there's at least one element g such that its powers (g^1, g^2, g^3, and so on) mod p span all distinct group elements. The element g is then a *generator* of the group. $\mathbf{Z}_5^*$ is cyclic and has two generators, 2 and 3, because $2^1 = 2$, $2^2 = 4$, $2^3 = 3$, $2^4 = 1$, and $3^1 = 3$, $3^2 = 4$, $3^3 = 2$, $3^4 = 1$.

Note that I'm using multiplication as a group operator, but you can also get groups from other operators. For example, the most straightforward group is the set of all integers, positive and negative, with addition as a group operation. Let's check that the group axioms hold with addition, in the preceding order: the number $x + y$ is an integer if x and y are integers (closure); $(x + y) + z = x + (y + z)$ for any x, y, and z (associativity); zero is the identity element; and the inverse of any number x in the group is $-x$ because $x + (-x) = 0$ for any integer x. A big difference, though, is that this group of integers is of infinite size, whereas in crypto you'll deal with only *finite groups*, or groups with a finite number of elements, for implementation reasons. Typically, you'll use groups $\mathbf{Z}_p^*$, where p is *thousands* of bits long (that is, groups that contain on the order of 2^m numbers if p is m-bit long).

The Hard Thing

The discrete logarithm problem as initially used in cryptography consists of finding the y for which $g^y = x$, given a generator g within some group $\mathbf{Z}_p^*$, where p is a prime number, and given a group element y. We often express the DLP in additive rather than multiplicative notation, as in groups of elliptic curves. In this case, the problem is to find the multiplicative factor k such that $k \times P = Q$, where you know the points P and Q. This is called the *elliptic curve DLP (ECDLP)*.

The DLP is *discrete* because you're dealing with countable integers as opposed to uncountable real numbers, and it's a *logarithm* because you're looking for the logarithm of x in base g. For example, the logarithm of 256 in base two is 8 because $2^8 = 256$.

Factoring is about as equally hard, thus as secure, as a discrete logarithm. In fact, algorithms to solve DLP bear similarities with those factoring integers, and you get about the same security level with n-bit hard-to-factor numbers as with discrete logarithms in an n-bit group. For the same reason as factoring, DLP isn't **NP**-hard. (There are certain groups where the DLP is easier to solve, but these aren't used in cryptography, at least not where DLP hardness is needed.)

How Things Can Go Wrong

More than 40 years later, we still don't know how to efficiently factor large numbers or solve discrete logarithms. In the absence of mathematical proof, it's always possible to speculate that they'll be broken one day. But we also don't have proof that **P** ≠ **NP**, so you can speculate that **P** may be equal to **NP**; however, according to experts, that surprise is unlikely. Most public-key crypto deployed today relies on either factoring (RSA) or DLP (Diffie–Hellman, ElGamal, elliptic curve cryptography). Although math may not fail us, real-world concerns and human error can sneak in.

When Factoring Is Easy

Factoring large numbers isn't always hard. For example, take the following 1,024-bit number N:

```
179769313486231590772930519078902473361797697894230657273430081157739343819933
842986982557174198257278917258638193709265819186026626180659730665062710995556
578639447715608415186895652841691982921107202317165369124890481512388558039053
427125099290315449262324709315263256083132540461407052872832790915388014592
```

For 1,024-bit numbers in RSA encryption or signature schemes where $N = pq$, we expect the best factoring algorithms to need around 2^{70} operations, as we discussed earlier. But you can factor this sample number in seconds using SageMath, a piece of Python-based mathematical software. Using SageMath's `factor()` function on my 2023 MacBook, it took less than a second to find the following factorization:

$$2^{800} \times 641 \times 6700417 \times 167773885276849215533569 \times 37414057161322375957408148834323969$$

Right, I cheated. This number isn't of the form $N = pq$ because it doesn't have just two large prime factors but rather five, including very small ones, which makes it easy to factor. First, you identify the $2^{800} \times 641 \times 6{,}700{,}417$ part by trying small primes from a precomputed list of prime numbers, which leaves you with a 192-bit number that's much easier to factor than a 1,024-bit number with two large factors.

Factoring can be easy not only when n has small prime factors but also when N or its factors p and q have particular forms—for example, when

$N = pq$ with p and q both close to some 2^b, when $N = pq$ and some bits of p or q are known, or when N is of the form $N = p^r q^s$ and r is greater than $\log p$. However, detailing the reasons for these weaknesses is too technical for this book.

The upshot is that the RSA encryption and signature algorithms (see Chapter 10) need to work with a value of $N = pq$, where p and q are carefully chosen, to avoid easy factorization of N, which can result in a security disaster.

Small Hard Problems Aren't Hard

Computationally hard problems, and even exponential-time algorithms, become practical when they're small enough. A symmetric cipher may be secure in the sense that there's no faster attack than the 2^n-time brute force, but if the key length is $n = 32$, you'll break the cipher in minutes. This sounds obvious, and you'd think that no one would be naive enough to use small keys, but in reality, there are plenty of reasons why this could happen. The following are two true stories.

Say you're a developer who knows nothing about crypto but has some API to encrypt with RSA and has been told to encrypt with 128-bit security. What RSA key size would you pick? I've seen real cases of 128-bit RSA, or RSA based on a 128-bit number $N = pq$. However, although factoring is impractically hard for an N thousands of bits long, factoring a 128-bit number is easy. Using the SageMath software, the commands in Listing 9-2 complete instantaneously.

```
sage: p = random_prime(2**64)
sage: print(p)
6822485253121677229
sage: q = random_prime(2**64)
sage: print(q)
17596998848870549923
Sage: N = p*q
sage: factor(N)
6822485253121677229 * 17596998848870549923
```

Listing 9-2: Generating an RSA modulus by picking two random prime numbers and factoring it instantaneously

Listing 9-2 shows that you can easily factor a 128-bit number taken randomly as the product of two 64-bit prime numbers on a typical laptop. However, if I chose 1,024-bit prime numbers instead by using `p = random_prime(2**1024)`, the command `factor(p*q)` would never complete, at least not in my lifetime.

To be fair, the tools available don't help prevent the naive use of insecurely short parameters. For example, the OpenSSL toolkit used to let you generate RSA keys as short as 31 bits without any warning; such short keys are totally insecure, as Listing 9-3 shows. OpenSSL has since been fixed, and its version 1.1.1t (from February 2023) returns an error "key size too small" if you request a key shorter than 512 bits.

```
$ openssl genrsa 31
Generating RSA private key, 31 bit long modulus
.++++++++++++++++++++++++++++
.++++++++++++++++++++++++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBHHqFuUCAwEAAQIEP6zEJQIDANATAgMAjCcCAwCSBwICTGsCAhpp
-----END RSA PRIVATE KEY-----
```

Listing 9-3: Generating an insecure RSA private key using an older version of the OpenSSL toolkit

When reviewing cryptography, you should check not only the type of algorithms used but also their parameters and the length of their secret values. However, as you'll see in the following story, what's secure enough today may be insecure tomorrow.

In 2015, researchers discovered that many HTTPS servers and email servers supported an older, insecure version of the Diffie–Hellman key agreement protocol. Namely, the underlying TLS implementation supported Diffie–Hellman within a group, $\mathbf{Z}_p^*$, defined by a prime number, p, of only 512 bits, where the discrete logarithm problem was no longer practically impossible to compute.

Not only did servers support a weak algorithm, but also attackers could force a benign client to use that algorithm by injecting malicious traffic within the client's session. Even better for attackers, the largest part of the attack could be carried out once and recycled to attack multiple clients. After about a week of computations to attack a specific group, $\mathbf{Z}_p^*$, it took only 70 seconds to break individual sessions of different users.

A secure protocol is worthless if it's undermined by a weakened algorithm, and a reliable algorithm is useless if sabotaged by weak parameters. In cryptography, you should always read the fine print.

For more details about this story, check the research article "Imperfect Forward Secrecy: How Diffie–Hellman Fails in Practice" (*https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf*).

Further Reading

I encourage you to look deeper into the foundational aspects of computation in the context of computability (what functions can be computed?) and complexity (at what cost?) and how they relate to cryptography. I've talked mostly about the classes **P** and **NP**, but there are many more classes and points of interest for cryptographers. I highly recommend the book *Quantum Computing Since Democritus* by Scott Aaronson (Cambridge University Press, 2013). It's in large part about quantum computing, but its first chapters brilliantly introduce complexity theory and cryptography.

In the cryptography research literature, you'll find other hard computational problems. I'll mention them in later chapters, but here are

some examples that illustrate the diversity of problems leveraged by cryptographers:

- The Diffie–Hellman problem (given g^x and g^y, find g^{xy}) is a variant of the discrete logarithm problem and is widely used in key agreement protocols.
- Lattice problems, such as the shortest vector problem (SVP) as well as the short integer solutions (SIS) and learning with errors (LWE) problems, can be **NP**-hard, depending on their parameters.
- Coding problems rely on the hardness of decoding error-correcting codes with insufficient information and have been studied since the late 1970s. These can also be **NP**-hard.
- Multivariate problems are about solving nonlinear systems of equations and are potentially **NP**-hard, but they've failed to provide reliable cryptosystems because hard versions are too big and slow, and practical versions were found to be insecure.

In Chapter 10, we'll continue exploring hard problems, especially factoring and its main variant, the RSA problem.

10

RSA

The Rivest–Shamir–Adleman (RSA) cryptosystem revolutionized cryptography when it emerged in 1977 as the first public-key encryption scheme. Whereas classical, symmetric-key encryption schemes use the same secret key to encrypt and decrypt messages, *public-key encryption* (or *asymmetric* encryption) uses two keys: a public key, which anyone who wants to encrypt messages for you can use, and a private key, which is required to decrypt messages encrypted using the public key. This magic is the reason why RSA was a real breakthrough, and 40 years later, it's still the paragon of public-key encryption and a workhorse of internet security. (One year prior to RSA, Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptography, but their scheme was able to perform key distribution only in a public-key setting.)

RSA works by creating a *trapdoor permutation*, a function that transforms a number x to a number y in the same range, such that computing y from x is easy using the public key, but computing x from y is practically impossible unless you know the private key—the trapdoor. (Think of x as a plaintext and y as a ciphertext.)

In addition to encryption, you can also use RSA to build digital signatures, wherein only the owner of the private key is able to sign a message, and the public key enables anyone to verify the signature's validity.

In this chapter, you'll learn how the RSA trapdoor permutation works and why this permutation alone isn't enough to build *secure* encryption and signatures. I also discuss RSA's security relative to the factoring problem (see Chapter 9), ways to implement RSA securely, and how to attack it.

We begin with an explanation of the basic mathematical notions behind RSA.

The Math Behind RSA

When processing a message, whether encrypting or signing it, RSA first creates a big number from that message and processes it by performing multiplications between big numbers. Therefore, to understand how RSA works, you need to know what kind of big numbers it manipulates and how multiplication works on those numbers.

To encrypt or sign, RSA transforms a positive integer between 1 and $n - 1$, where n is a large number called the *modulus*. Such numbers, when multiplied together, yield another number that satisfies these criteria. These numbers form a group, which you denote $\mathbf{Z}_n^*$ and call the multiplicative group of integers modulo n. (See the mathematical definition of a group in "Groups" on page 189.)

For example, consider the group $\mathbf{Z}_4^*$ of integers modulo 4. Recall from Chapter 9 that a group must include an identity element (that is, 1) and that each number x in the group must have an inverse, a number y such that $x \times y = 1$. How do you determine the set that makes up $\mathbf{Z}_4^*$? Based on the definitions, you know that 0 is not in the group $\mathbf{Z}_4^*$ because multiplying any number by 0 can never give 1, so 0 has no inverse. The number 1 belongs to $\mathbf{Z}_4^*$ because $1 \times 1 = 1$, so 1 is its own inverse. However, the number 2 doesn't belong in this group because you can't obtain 1 by multiplying 2 with another element of $\mathbf{Z}_4^*$ (note that 2 isn't coprime with 4, because 4 and 2 share the factor of 2). The number 3 belongs in the group $\mathbf{Z}_4^*$ because it is its own inverse within $\mathbf{Z}_4^*$. Thus, you have $\mathbf{Z}_4^* = \{1, 3\}$.

Now consider $\mathbf{Z}_5^*$, the multiplicative group of integers modulo 5. As in Chapter 9, $\mathbf{Z}_5^* = \{1, 2, 3, 4\}$. Indeed, 5 being prime, 1, 2, 3, and 4 are all coprime with 5, so the set of $\mathbf{Z}_5^*$ includes all of them. Let's verify this: $2 \times 3 \bmod 5 = 1$; therefore, 2 is 3's inverse, and 3 is 2's inverse; note that 4 is its own inverse because $4 \times 4 \bmod 5 = 1$; and finally, 1 is again its own inverse in the group.

To find the number of elements in a group $\mathbf{Z}_n^*$ when n isn't prime, you can use *Euler's totient function*, which we write as $\varphi(n)$, with φ representing

the Greek letter phi. This function gives the number of elements coprime with n, which is the number of elements in $\mathbf{Z}_n^*$. As a rule, if n is a product of prime numbers $n = p_1 \times p_2 \times \ldots \times p_m$, the number of elements in the group $\mathbf{Z}_n^*$ is the following:

$$\varphi(n) = (p_1 - 1) \times (p_2 - 1) \times \ldots \times (p_m - 1)$$

RSA deals only with numbers n that are the product of two large primes, usually noted as $n = pq$. The associated group $\mathbf{Z}_n^*$ then contains $\varphi(n) = (p-1)(q-1)$ elements. By expanding this expression, you get the equivalent definition $\varphi(n) = n - p - q + 1$, or $\varphi(n) = (n+1) - (p+q)$, which expresses more intuitively the value of $\varphi(n)$ relative to n.

The RSA Trapdoor Permutation

The RSA trapdoor permutation is the core algorithm behind RSA-based encryption and signatures. Given a modulus n and a number e, which you call the *public exponent*, the RSA trapdoor permutation transforms a number x from the set $\mathbf{Z}_n^*$ into a number $y = x^e \bmod n$. In other words, it calculates the value that's equal to x multiplied by itself $e-1$ times modulo n and then returns the result. When you use the RSA trapdoor permutation to encrypt, the modulus n and the exponent e make up the RSA public key.

To get x back from y, you can use another number, d, to compute the following:

$$y^d \bmod n = \left(x^e\right)^d \bmod n = x^{ed} \bmod n = x$$

Because d is the trapdoor that allows you to decrypt, it's part of the private key in an RSA key pair, meaning it should always be kept secret. The number d is also called the *secret exponent*.

Of course, d isn't just any number; it's the number such that e multiplied by d is equivalent to 1 and therefore such that $x^{ed} \bmod n = x$ for any x. More precisely, you must have $ed \bmod \varphi(n) = 1$ to get $x^{ed} = x^1 = x$ and to decrypt the message correctly. Note that you compute modulo $\varphi(n)$ and not modulo n here because exponents behave like *indexes* of elements of $\mathbf{Z}_n^*$ rather than as the elements themselves. Because $\mathbf{Z}_n^*$ has $\varphi(n)$ elements, the index must be less than $\varphi(n)$.

The number $\varphi(n)$ is crucial to RSA's security. In fact, finding $\varphi(n)$ for an RSA modulus n is equivalent to breaking RSA because you can easily derive the secret exponent d from $\varphi(n)$ and e by computing e's inverse. Hence p and q should also be secret, since knowing p or q gives $\varphi(n)$ by computing $(p-1)(q-1) = \varphi(n)$.

NOTE *$\varphi(n)$ is called the* order *of the group $\mathbf{Z}_n^*$; the order is an important characteristic of a group, which is essential to other public-key systems such as Diffie–Hellman and elliptic curve cryptography.*

RSA Key Generation and Security

Key generation is the process by which an RSA key pair is created, namely, a public key (modulus n and public exponent e) and its private key (secret exponent d). As the numbers p and q (such that $n = pq$) and the order $\varphi(n)$ should also be secret, you will often include them as part of the private key.

To generate an RSA key pair, first pick two random prime numbers, p and q, and compute $\varphi(n)$ from them. Then compute d as the inverse of e. Listing 10-1 shows how this works using SageMath (*https://www.sagemath.org*), an open source Python-esque environment that includes many mathematical packages.

```
sage: p = random_prime(2^32); p
1103222539
sage: q = random_prime(2^32); q
17870599
sage: n = p*q; n
19715247602230861
sage: phi = (p-1)*(q-1); phi
19715246481137724
sage: e = random_prime(phi); e
13771927877214701
sage: d = xgcd(e, phi)[1]; d = mod(d, phi)
11417851791646385
sage: mod(d*e, phi)
1
```

Listing 10-1: Generating RSA parameters using SageMath

Here you use the `random_prime()` function to pick random primes `p` and `q`, which are lower than a given argument. Next, you multiply `p` and `q` to get the modulus `n` and $\varphi(n)$, which is the `phi` variable. You then generate a random public exponent, `e`, by picking a random prime less than `phi` to ensure that `e` has an inverse modulo `phi`.

You use the `xgcd()` function from Sage to generate the associated private exponent d. This function computes the numbers s and t given two numbers, a and b, with the extended Euclidean algorithm such that $as + bt = \mathbf{GCD}(a, b)$. Finally, you check that $ed \bmod \varphi(n) = 1$ to ensure that d correctly inverts the RSA permutation.

NOTE *I've used a 64-bit modulus* `n` *in Listing 10-1 to avoid multiple pages of output, but in practice an RSA modulus should be at least 2,048 bits to be secure.*

Now you can apply the trapdoor permutation, as Listing 10-2 shows.

```
sage: x = 1234567
sage: y = power_mod(x, e, n); y
17129109575774132
sage: power_mod(y, d, n)
1234567
```

Listing 10-2: Computing the RSA trapdoor permutation back and forth

You assign the integer 1,234,567 to x and then use the `power_mod(x, e, n)` function, the exponentiation modulo n, or $x^e \bmod n$ in equation form, to calculate y. After computing $y = x^e \bmod n$, you calculate $y^d \bmod n$ with the trapdoor d to return the original x.

How hard is it to find x without the trapdoor d? An attacker able to factor big numbers could break RSA by recovering p and q and then $\varphi(n)$ to compute d from e. Another risk to RSA lies in an attacker's ability to compute x from $x^e \bmod n$, or eth roots modulo n, without necessarily factoring n. While both risks seem closely connected, we don't know for sure that they're equivalent.

Assuming that factoring and finding eth roots are about equally hard, RSA's security level depends on three factors: the size of n, the choice of p and q, and how the trapdoor permutation is used. If n is too small, one could factor it in a realistic amount of time, revealing the private key. To be safe, n should be at least 2,048 bits long (a security level of about 90 bits, requiring a computational effort of about 2^{90} operations) but preferably 4,096 bits long (a security level of approximately 128 bits). The values p and q should be unrelated random prime numbers of similar size. If they're too small or too close together, it's easier to determine their value from n. Finally, you shouldn't use the RSA trapdoor permutation directly for encryption or signing, as I'll discuss shortly.

Encrypting with RSA

Typically, RSA is used in combination with a symmetric encryption scheme, where RSA encrypts a symmetric key that serves to encrypt a message with a symmetric cipher, such as AES-GCM. But encrypting a message or symmetric key with RSA is more complicated than simply converting the target to a number x and computing $x^e \bmod n$.

In the following subsections, I explain why a naive application of the RSA trapdoor permutation is insecure and how strong RSA-based encryption works.

Textbook RSA Encryption's Malleability

The phrase *textbook RSA encryption* describes the simplistic RSA encryption scheme wherein the number exponentiated contains only the message you want to encrypt. For example, to encrypt the string *RSA*, you first convert it to a number—for example, by concatenating the ASCII encodings of each of

the three letters as a byte: *R* (byte 52), *S* (byte 53), and *A* (byte 41). The resulting byte string 525341 is equal to 5,395,265 when converted to decimal, which you might then encrypt by computing $5{,}395{,}265^e \bmod n$. Without knowing the secret key, there's no way to decrypt the message—at least in theory.

However, textbook RSA encryption is deterministic: if you encrypt the same plaintext twice, you'll get the same ciphertext twice. And there's an even bigger problem: given two textbook RSA ciphertexts $y_1 = x_1^e \bmod n$ and $y_2 = x_2^e \bmod n$, you can derive the ciphertext of $x_1 \times x_2$ by multiplying these ciphertexts together:

$$y_1 \times y_2 \bmod n = x_1^e \times x_2^e \bmod n = (x_1 \times x_2)^e \bmod n$$

The result is $(x_1 \times x_2)^e \bmod n$, the ciphertext of the message $x_1 \times x_2 \bmod n$. An attacker could create a new valid ciphertext from two RSA ciphertexts, compromising the security of your encryption by allowing them to deduce information about the original message. This weakness makes textbook RSA encryption *malleable*. (If you know x_1 and x_2, you can compute $(x_1 \times x_2)^e \bmod n$, too, but if you know only y_1 and y_2, you shouldn't be able to multiply ciphertexts and get the ciphertext of the multiplied plaintexts.)

Another problem with the naive textbook RSA encryption is the existence of "special" messages. Whatever n and e are, $1^e = 1$. Thus, the message 1 is left unchanged through encryption. Textbook RSA has many other problems, but you'll learn to avoid them by using a strong RSA encryption.

Strong RSA Encryption with OAEP

To make RSA ciphertexts nonmalleable, the number exponentiated during encryption should combine message data with additional data called *padding*, as Figure 10-1 shows.

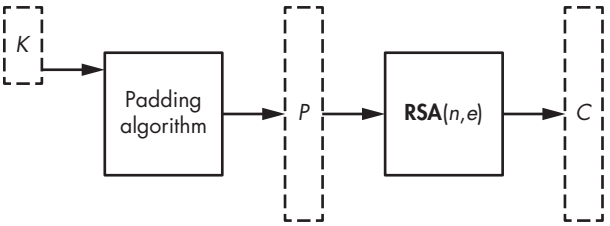


Figure 10-1: Encrypting a symmetric key, K, with RSA using (n, e) as a public key

The standard way to encrypt with RSA in this fashion is to use Optimal Asymmetric Encryption Padding (OAEP), a combination commonly called RSA-OAEP. This scheme involves creating a bit string as large as the modulus by padding the message with extra data and randomness before applying the RSA function.

NOTE *Official documents such as the PKCS#1 standard and NIST's Special Publication 800-56B refer to OAEP as RSAES-OAEP. OAEP improves on the earlier method*

PKCS#1 v1.5, which is one of the first in a series of Public-Key Cryptography Standards (PKCS) by RSA. It is markedly less secure than OAEP yet has remained in use in many systems after OAEP's introduction.

Security

OAEP uses a PRNG to ensure the indistinguishability and nonmalleability of ciphertexts by making the encryption probabilistic. It's been proven secure as long as the RSA function and the PRNG are secure and, to a lesser extent, the hash functions aren't too weak. You should use OAEP when encrypting with RSA, rather than its predecessor, the standard PKCS#1 v1.5.

Encryption

Encrypting with RSA in OAEP mode requires a message (such as a symmetric key, K), a PRNG, and two hash functions. To create the ciphertext, use a given modulus n long of m bytes (that is, $8m$ bits and therefore an n lower than 2^{8m}). To encrypt K, form the *encoded message* as

$$M = H \,||\, 00 \ldots 00 \,||\, 01 \,||\, K$$

where H is an h-byte constant defined by the OAEP scheme, followed by as many 00 bytes as necessary and a 01 byte. Figure 10-2 shows how to process this encoded message, M.

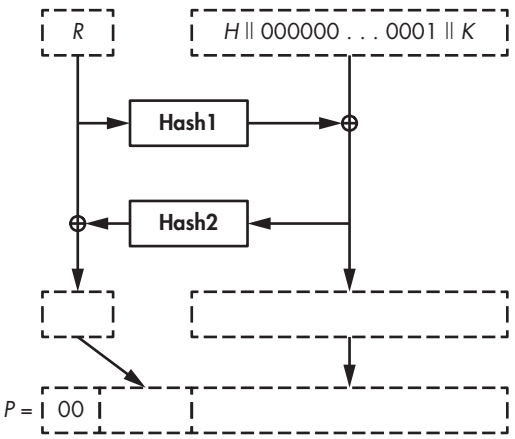


Figure 10-2: Encrypting a symmetric key, K, with RSA-OAEP, where H is a fixed parameter and R is random bits

You generate an h-byte random string R and set $M = M \oplus \mathbf{Hash1}(R)$, where $\mathbf{Hash1}(R)$ is as long as M. Next, set $R = R \oplus \mathbf{Hash2}(M)$, where $\mathbf{Hash2}(M)$ is as long as R. Now use these new values of M and R to form an m-byte string $P = 00 \,||\, M \,||\, R$, which is as long as the modulus n and which

you can convert to an integer number less than n. This conversion results in the number x, which you then use to compute the RSA function $x^e \bmod n$ to get the ciphertext.

To decrypt a ciphertext y, first compute $x = y^d \bmod n$ and, from this, recover the final values of M and R. Next, retrieve M's initial value by calculating $M \oplus \textbf{Hash1}(R \oplus \textbf{Hash2}(M))$. Finally, verify that M is of the form $H \,\|\, \texttt{00} \ldots \texttt{00} \,\|\, \texttt{01} \,\|\, K$, with an h-byte H and `00` bytes followed by a `01` byte.

In practice, the parameters m and h (the length of the modulus and the length of **Hash2**'s output, respectively) are typically $m = 256$ bytes (for 2,048-bit RSA) and $h = 32$ (using SHA-256 as **Hash2**). This leaves $m - h - 1 = 223$ bytes for M, of which up to $m - 2h - 2 = 190$ bytes are available for K (the "– 2" is due to the `01` byte separator in M). The **Hash1** hash value is then composed of $m - h - 1 = 223$ bytes, which is longer than the hash value of any common hash function. To build a hash with such an unusual output length, the RSA standard specification defines a *mask generating function* technique to create hash functions that return arbitrarily large hash values from any hash function. As another approach, use an extendable output function (XOF) such as SHAKE or BLAKE3, although this differs from the standard specifications.

Signing with RSA

Digital signatures can prove that the holder of the private key tied to a particular digital signature signed some message, typically to endorse its content. Because only the private key holder knows the private exponent d, no one else can compute a signature $y = x^d \bmod n$ from some value x, but everyone can verify $y^e \bmod n = x$ given the public exponent e. In principle, one can use that verified signature as evidence to demonstrate that the private-key holder signed some particular message; this is a property called *nonrepudiation*.

It's tempting to see RSA signatures as the converse of encryption, but they aren't. Signing with RSA isn't the same as encrypting with the private key. Encryption provides confidentiality, whereas digital signatures help prevent forgeries. The most salient example of this difference is that it's OK for a signature scheme to leak information on the message signed, because the message isn't secret. For example, a scheme that reveals parts of the messages could be a secure signature scheme but not a secure encryption scheme.

Because of the necessary processing overhead, public-key encryption can process only short messages, which are usually secret keys rather than actual messages. A signature scheme, however, can process messages of arbitrary sizes by using their hash values $\textbf{Hash}(M)$ as a proxy, and it can be deterministic yet secure. Like RSA-OAEP, RSA-based signature schemes can use a padding scheme, but they can also use the maximal message space allowed by the RSA modulus.

Textbook RSA Signatures

A *textbook RSA signature* is the method that signs a message, x, by directly computing $y = x^d \bmod n$, where x can be any number between 1 and $n - 1$. Like textbook encryption, textbook RSA signing is simple to specify and implement but insecure in the face of several attacks. One such attack involves a trivial forgery: upon noticing that $1^d \bmod n = 1$ and $(n - 1)^d \bmod n = n - 1$, regardless of the value of the private key d, an attacker can forge signatures of 1 or $n - 1$ without knowing d.

More worrying is the *blinding attack*. For example, say you want to get a third party's signature on some message, M, that you know they'd never knowingly sign. To launch this attack, first find some value, R, such that $R^eM \bmod n$ is a message that your victim would knowingly sign. Next, you'd convince them to sign that message and to show you their signature, which is equal to $S = (R^eM)^d \bmod n$, or the message raised to the power d. Given that signature, you can derive the signature of M, namely, M^d, with the aid of some straightforward computations.

Because you can write S as $(R^eM)^d = R^{ed}M^d$, and because $R^{ed} = R$ (by definition), you have $S = (R^eM)^d = RM^d$. To obtain the signature M^d, divide S by R as follows:

$$S / R = RM^d / R = M^d$$

This is often a practical and powerful attack.

The PSS Signature Standard

The RSA *Probabilistic Signature Scheme (PSS)* is to RSA signatures what OAEP is to RSA encryption. It was designed to make message signing more secure, thanks to the addition of padding data.

Figure 10-3 demonstrates that PSS combines a message narrower than the modulus with some random and fixed bits before RSAing the results of this padding process.

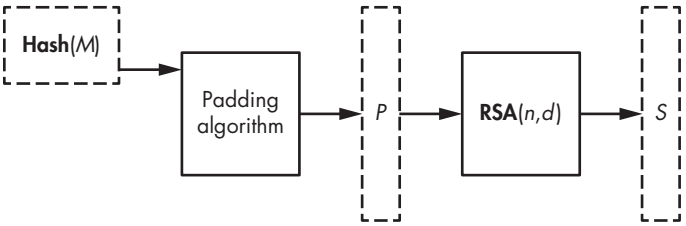


Figure 10-3: Signing a message, M, with RSA and with the PSS standard, where (n, d) is the private key

Like all public-key signature schemes, PSS works on a message's hash rather than on the message itself. Signing **Hash**(M) is secure only if the hash function is collision resistant. You can thus sign messages of any

length because after hashing a message, you'll obtain a hash value of the same fixed length regardless of the message's original length.

Why not sign by just applying the OAEP encryption to **Hash**(M)? Unfortunately, you can't. Although similar to PSS, OAEP has been proven secure only for encryption, not for signatures.

Like OAEP, PSS also requires a PRNG and two hash functions. One, **Hash1**, is a typical hash with a standard output length, such as SHA-256. The other, **Hash2**, is a wide-output hash like OAEP's **Hash2**. Like OAEP, PSS can use the mask-generating function (MGF) construction to build such a hash.

The PSS signing procedure works as follows (where h is **Hash1**'s output length):

1. Pick an r-byte random string R using the PRNG.
2. Form an encoded message $M' = \texttt{0000000000000000} \,||\, \textbf{Hash1}(M) \,||\, R$, long of $h + r + 8$ bytes (with 8 zero bytes at the beginning).
3. Compute the h-byte string $H = \textbf{Hash1}(M')$.
4. Set $L = \texttt{00...00} \,||\, \texttt{01} \,||\, R$, or a sequence of `00` bytes followed by a `01` byte and then R, with a number of `00` bytes such that L is long of $m - h - 1$ bytes (the byte width m of the modulus minus the hash length h minus 1).
5. Set $L = L \oplus \textbf{Hash2}(H)$ to replace the previous value of L with a new value.
6. Convert the m-byte string $P = L \,||\, H \,||\, \texttt{BC}$ to a number, x, lower than n. Here, the byte `BC` is a fixed value appended after H.
7. Given the value of x just obtained, compute the RSA function $x^d \bmod n$ to obtain the signature.
8. To verify a signature given a message, M, compute **Hash1**(M) and use the public exponent e to inverse the RSA function and retrieve L, H, and then M' from the signature, checking the padding's correctness at each step.

In practice, the random string R (called a *salt* t in the RSA-PSS standard) is usually as long as the hash value. For example, if you use $n = 2{,}048$ bits and SHA-256 as the hash, the value L is $m - h - 1 = 256 - 32 - 1 = 223$ bytes, and the random string R would typically be 32 bytes.

Like OAEP, PSS is provably secure, standardized, and available in many cryptographic software utilities and libraries, including OpenSSL and the Go language's cryptography module. Also like OAEP, it looks needlessly complex and is prone to implementation errors and mishandled corner cases. Unlike RSA encryption, there's a simpler alternative to PSS for signing: without a PRNG, with a single hash function, and without padding.

Full Domain Hash Signatures

Full Domain Hash (FDH) is the simplest signature scheme you can imagine. To implement it, convert the byte string **Hash**(M) to a number, x, and create the signature $y = x^d \bmod n$, as in Figure 10-4.

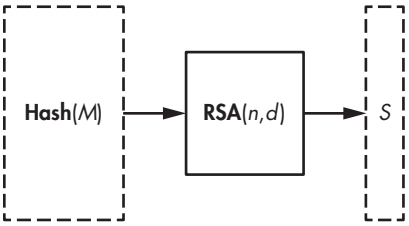


Figure 10-4: Signing a message with RSA using the Full Domain Hash technique

Signature verification is straightforward, too. Given a signature that's a number y, compute $x = y^e \bmod n$ and compare the result with **Hash**(M). It's boringly simple, deterministic, yet secure. So why bother with the complexity of PSS?

PSS was released *after* FDH, in 1996, and it has a security proof that inspires more confidence than FDH. Specifically, its proof offers slightly higher security guarantees than the proof of FDH, and its use of randomness helped strengthen that proof.

These stronger theoretical guarantees are the main reason why many cryptographers prefer PSS over FDH, but most applications using PSS today could switch to FDH with no meaningful security loss. In some contexts, however, it's viable to use PSS instead of FDH because PSS's randomness protects it from some attacks on its implementation, such as the fault attacks we'll discuss in "How Things Can Go Wrong" on page 211.

In any case, RSA signatures (PSS or FDH) are less and less used. Signatures based on elliptic curves, such as ECDSA and EdDSA, have gained in popularity, not least because their calculation of a signature is much faster (although signature verification is often faster with RSA).

RSA Implementations

I hope you'll never have to implement RSA from scratch. If you're asked to, run as fast as you can and question the sanity of the person who asked you to do so. It took decades for cryptographers and engineers to develop RSA implementations that are fast, sufficiently secure, and ideally free of debilitating bugs, so it isn't wise to reinvent RSA. Even with all the documentation available, it would take months to complete this daunting task.

Typically, when using RSA in software, you'll use a library or API that provides the necessary functions to carry out RSA operations. For example,

the Go language has the following function in its `crypto` package (from *https://go.dev/src/crypto/rsa/rsa.go*):

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte,
label []byte) (out []byte, err error)
```

The function `EncryptOAEP()` takes a hash function, a PRNG, a public key, a message, and a label (an optional parameter of OAEP) and returns a ciphertext and an error code. When you call `EncryptOAEP()`, it calls `encrypt()` to compute the RSA function given the padded data, as Listing 10-3 shows.

```
func encrypt(c *big.Int, pub *PublicKey, m *big.Int) *big.Int {
    e := big.NewInt(int64(pub.E))
    c.Exp(m, e, pub.N)
    return c
}
```

Listing 10-3: The implementation of the core RSA encryption function from the Go language cryptography library

The main operation here is `c.Exp(m, e, pub.N)`, which raises a message, `m`, to the power `e` modulo `pub.N`, and assigns the result to the variable `c`.

If you choose to implement RSA instead of using a readily available library function, be sure to rely on an existing *big-number* library, which is a set of functions and types that allow you to define and compute arithmetic operations on large numbers thousands of bits long. For example, you might use the GNU Multiple Precision (GMP) arithmetic library in C or in Go's `big` package. (Believe me, you don't want to implement big-number arithmetic yourself.)

Even if you just use a library function when implementing RSA, ensure you understand how the internals work so you can assess the risk and whether it matches the application's security requirements.

A Fast Exponentiation Algorithm

Exponentiation is the operation of raising x to the power e, when computing $x^e \bmod n$. When working with big numbers, as with RSA, this operation can be extremely slow if naively implemented. How to efficiently compute exponentiations?

The naive way to compute $x^e \bmod n$ takes $e - 1$ multiplications, as the pseudocode algorithm in Listing 10-4 demonstrates.

```
expModNaive(x, e, n) {
    y = x
    for i = 1 to e - 1 {
        y = y * x mod n
    }
```

```
    return y
}
```

Listing 10-4: A naive exponentiation algorithm in pseudocode, raising x *to the power* e *modulo* n

This algorithm is simple but highly inefficient. You can get the same result exponentially faster by squaring rather than multiplying intermediate values y until you reach the correct value. This family of methods is called *square-and-multiply*, *exponentiation by squaring*, or *binary exponentiation*.

Say you want to compute $3^{65,537}$ mod 36,567,232,109,354,321, for example. (The number 65,537 is the public exponent in most RSA implementations.) You could multiply the number 3 by itself 65,536 times, or you could approach this problem knowing that you can write 65,537 as $2^{16} + 1$ and use a series of squaring operations instead. Essentially, you initialize a variable, $y = 3$, and then compute the following squaring (y^2) operations:

1. Set $y = y^2$ mod n (now $y = 3^2$ mod n).
2. Set $y = y^2$ mod n (now $y = (3^2)^2$ mod $n = 3^4$ mod n).
3. Set $y = y^2$ mod n (now $y = (3^4)^2 = 3^8$ mod n).
4. Set $y = y^2$ mod n (now $y = (3^8)^2 = 3^{16}$ mod n).
5. Set $y = y^2$ mod n (now $y = (3^{16})^2 = 3^{32}$ mod n).

And so on until $y = 3^{65,536}$, by performing 16 squarings.

To get the final result, you would return $3 \times y$ mod $n = 3^{65,537}$ mod n = 26,652,909,283,612,267. In other words, you compute the result with only 17 multiplications rather than 65,536 with the naive method.

More generally, a square-and-multiply method works by scanning the exponent's bits one by one, computing the square for each exponent's bit to double the exponent's value, and multiplying by the original number for each bit with a value of 1 encountered. In the preceding example, the exponent 65,537 is 10000000000000001 in binary, and you squared y for each new bit and multiplied by the original number 3 only for the very first and last bits.

Listing 10-5 shows how this works as a general algorithm in pseudocode to compute x^e mod n when the exponent e consists of bits $e_{m-1}e_{m-2} \cdots e_1e_0$, where e_0 is the least significant bit.

```
expMod(x, e, n) {
    y = x
    for i = m - 1 to 0 {
        y = y * y  mod n
        if e_i == 1 then
            y = y * x  mod n
    }
    return y
}
```

Listing 10-5: A fast exponentiation algorithm in pseudocode, raising x *to the power* e *modulo* n

The `expMod()` algorithm runs in time $O(m)$, whereas the naive algorithm runs in time $O(2^m)$, where m is the bit length of the exponent. Here, $O()$ is the asymptotic complexity notation introduced in Chapter 9.

All serious systems implement some variant of this simplest square-and-multiply method. One such variant is the *sliding window* method, which considers blocks of bits rather than individual bits to perform a given multiplication operation. For example, see the function `expNN()` of the Go language, whose source code is available at *https://go.dev/src/math/big/nat.go*.

How secure are these square-and-multiply exponentiation algorithms? Unfortunately, the tricks to speed the process up often result in increased vulnerability against some attacks.

The weakness of these algorithms stems from the fact that the exponentiation operations are heavily dependent on the exponent's value. The `if` operation in Listing 10-5 takes a different branch based on whether an exponent's bit is 0 or 1. If a bit is 1, an iteration of the `for` loop is slower than it is for 0, and attackers who monitor the execution time of the RSA operation can exploit this time difference to recover a private exponent. This is called a *timing attack*. Attacks on hardware can distinguish 1 bit from 0 bits by monitoring the device's power consumption and observing which iterations perform an extra multiplication to reveal which bits of the private exponent are 1.

Depending on the type of platform, the information channel may not be execution time. For example, you might distinguish 1-bits in the exponent from 0-bits by measuring the device's power consumption and observing which iterations perform additional multiplication, revealing the private exponent bits at 1. This is a *power-analysis attack*. Few open source cryptographic libraries contain effective defenses against these types of attacks.

Small Exponents for Faster Public-Key Operations

Because an RSA computation is essentially the computation of an exponentiation, its performance depends on the value of the numbers involved, in particular the exponent. Smaller exponents require fewer multiplications and can therefore make the exponentiation computation much faster.

In principle, the public exponent e can be any value between 3 and $\varphi(n) - 1$, as long as e and $\varphi(n)$ are coprime. But in practice you'll find only small values of e, and most of the time $e = 65{,}537$ due to concerns with encryption and signature verification speed. For example, the Microsoft Windows CryptoAPI supports only public exponents that fit in a 32-bit integer. The larger the e, the slower it is to compute $x^e \bmod n$.

Unlike the size of the public exponent, the private exponent d is inevitably about as large as n, making decryption much slower than encryption, and signing much slower than verification. Because d is secret, it must be unpredictable and therefore can't be restricted to a small value. For example, if e is

fixed to 65,537, the corresponding d is usually of the same order of magnitude as the modulus n, which would be close to $2^{2,048}$ if n is 2,048 bits long.

As discussed in "A Fast Exponentiation Algorithm" on page 206, raising a number to the power 65,537 takes 17 multiplications, whereas raising a number to the power of some 2,048-bit number takes on the order of 3,000 multiplications.

You can estimate the speed of RSA by using the OpenSSL toolkit. For example, Listing 10-6 shows the results of 512-, 1,024-, 2,048-, and 4,096-bit RSA operations on a MacBook equipped with an M2 chipset:

```
Doing 1024 bits private rsa sign ops for 10s: 119019 1024 bits private RSA sign ops in 9.93s
Doing 1024 bits public rsa verify ops for 10s: 2573979 1024 bits public RSA verify ops in 9.93s
Doing 2048 bits private rsa sign ops for 10s: 19184 2048 bits private RSA sign ops in 9.93s
Doing 2048 bits public rsa verify ops for 10s: 769696 2048 bits public RSA verify ops in 9.93s
Doing 4096 bits private rsa sign ops for 10s: 3005 4096 bits private RSA sign ops in 9.92s
Doing 4096 bits public rsa verify ops for 10s: 206367 4096 bits public RSA verify ops in 9.91s

                  sign    verify    sign/s verify/s
rsa 1024 bits 0.000083s 0.000004s 11985.8 259212.4
rsa 2048 bits 0.000518s 0.000013s  1931.9  77512.2
rsa 4096 bits 0.003301s 0.000048s   302.9  20824.1
--snip--
```

Listing 10-6: Example benchmarks of RSA operations using the OpenSSL toolkit (version 3.2.0)

To get an idea of how much slower verification is compared to signature generation, compute the ratio of the verification time over signature time. The benchmarks in Listing 10-6 show that I've got verification-over-signature speed ratios of approximately 21.6, 40.1, and 68.7 for 1,024-, 2,048-, and 4,096-bit moduli, respectively. The gap grows with the modulus size because the number of multiplications for e operations remains constant with respect to the modulus size (for example, 17 operations when $e = 65,537$), while private-key operations always need more multiplications for a greater modulus because d grows accordingly.

If small exponents are so nice, why use 65,537 and not something like 3? It would actually be fine (and faster) to use 3 as an exponent when implementing RSA with a secure scheme such as OAEP, PSS, or FDH. Cryptographers avoid doing so, however, because when $e = 3$, less secure schemes make certain types of mathematical attacks possible. The number 65,537 is large enough to avoid such *low-exponent attacks*, and it has only two nonzero bits, which decreases the computation time when computing $x^{65,537}$. The number 65,537 is also special for mathematicians: it's the fourth Fermat number, or a number of the form

$$2^{(2^n)} + 1$$

because it's equal to $2^{16} + 1$, where $16 = 2^4$, but that's a curiosity irrelevant to most cryptographic engineers.

The Chinese Remainder Theorem

The most common trick to speed up decryption and signature generation (that is, the computation of $y^d \bmod n$) is the *Chinese remainder theorem (CRT)*, which makes RSA about four times faster.

The CRT allows for faster decryption by computing two exponentiations, modulo p and modulo q, rather than a single one modulo n. Because p and q are much smaller than n, it's faster to perform two "small" exponentiations than a single "big" one.

The Chinese remainder theorem isn't specific to RSA. It's a general arithmetic result that, in its simplest form, states that if $n = n_1 n_2 n_3 \ldots$, where the n_is are pairwise coprime (that is, $\mathbf{GCD}(n_i, n_j) = 1$ for any distinct i and j), then one can compute the value $x \bmod n$ from the values $x \bmod n_1$, $x \bmod n_2$, $x \bmod n_3$, . . . For example, say you have $n = 1{,}155$, which is the product of prime factors $3 \times 5 \times 7 \times 11$, and say you want to determine x that satisfies $x \bmod 3 = 2$, $x \bmod 5 = 1$, $x \bmod 7 = 6$, and $x \bmod 11 = 8$. (I've chosen 2, 1, 6, and 8 arbitrarily.) To find x using the Chinese remainder theorem, compute the sum $P(n_1) + P(n_2) + \ldots$, where $P(n_i)$ is defined as follows:

$$P(n_i) = (x \bmod n_i) \times n/n_i \times (1/(n/n_i) \bmod n_i) \bmod n$$

Note that the second term, n/n_i, is equal to the product of all factors other than this n_i.

To apply this formula to this example and recover $x \bmod 1155$, compute $P(3)$, $P(5)$, $P(7)$, and $P(11)$; then add them together to get the following expression:

$$\left[\begin{array}{l} 2 \times 385 \times (1/385 \bmod 3) + 1 \times 231 \times (1/231 \bmod 5) + 6 \\ \times 165 \times (1/165 \bmod 7) + 8 \times 105 \times (1/105 \bmod 11) \end{array}\right] \bmod n$$

Here, I've just applied the preceding definition of $P(n_i)$. (The math behind finding each number is straightforward, but I won't detail it here.) You can then reduce this expression to $[770 + 231 + 1980 + 1680] \bmod n = 41$. As 41 is the number I'd picked for this example, you've got the correct result.

Applying the CRT to RSA is simpler than the previous example because there are only two factors for each n (namely, p and q). Given a ciphertext y to decrypt, instead of computing $y^d \bmod n$, use the CRT to compute $x_p = y^s \bmod p$, where $s = d \bmod (p - 1)$ and $x_q = y^t \bmod q$, where $t = d \bmod (q - 1)$. Combine these two expressions and compute x to be the following:

$$x = (x_p \times q \times (1/q \bmod p) + x_q \times p \times (1/p \bmod q)) \bmod n$$

That's it. This is faster than a single exponentiation even with the square-and-multiply trick because the multiplication-heavy operations are carried out on modulo p and q, numbers that are twice as small as n, whereas the complexity of exponentiation increases faster than linearly. When you double the size of the number, you don't just double the cost of exponentiation—it grows faster.

NOTE *In the final operation, you can compute the two numbers* q × *(1/*q *mod* p*) and* p × *(1/*p *mod* q*) in advance, which means you need to compute only two multiplications and an addition of modulo* n *to find* x.

Unfortunately, there's a security caveat attached to these techniques.

How Things Can Go Wrong

Even more elegant than the RSA scheme itself is the range of attacks that work either because the implementation leaks (or can be made to leak) information on its internals or because RSA is used insecurely. I discuss two classic examples of these types of attacks in the sections that follow.

The Bellcore Attack on RSA-CRT

The Bellcore attack on RSA is one of the most important attacks in the history of RSA. First discovered in 1996, it stood out because it exploited RSA's vulnerability to *fault injections*—attacks that force part of the algorithm to misbehave and potentially yield incorrect results. For example, one may temporarily perturb hardware circuits or embedded systems by suddenly altering their voltage supply or beaming a laser pulse to a carefully chosen part of a chip. Attackers can then exploit the resulting faults in an algorithm's internals by observing the impact on the result. Comparing the correct result with a faulty one can provide information on the algorithm's internal values, including secret values. Likewise, attempts to learn whether the result is valid might leak exploitable information.

The Bellcore attack is such a fault attack. It works on RSA signature schemes that use the Chinese remainder theorem and that are deterministic—meaning it works on FDH but not on PSS, which is probabilistic.

To understand the Bellcore attack, recall from the previous section that with CRT, you obtain the result that's equal to $x^d \bmod n$ by computing the following, where $x_p = y^s \bmod p$ and $x_q = y^t \bmod q$:

$$x = x_p \times q \times \left(1/q \bmod p\right) + x_q \times p \times \left(1/p \bmod q\right) \bmod n$$

Now assume that an attacker induces a fault in the computation of x_q so that you end up with some incorrect value, which differs from the actual x_q. Let's call this incorrect value x_q' and the final result obtained x'. The attacker can then subtract the incorrect signature x' from the correct signature x to factor n, which results in the following:

$$x - x' = \left(x_q - x_q'\right) \times p \times \left(1/p \bmod q\right) \bmod n$$

The value $x - x'$ is therefore a multiple of p, so p is a divisor of $x - x'$. Because p is also a divisor of n, the greatest common divisor of n and $x - x'$ yields p, $\mathbf{GCD}(x - x', n) = p$. You can then compute $q = n/p$ and d, resulting in a total break of RSA signatures.

A variant of this attack works when you know only that the message is signed, and not the correct signature. There's also a similar fault attack on the modulus value, rather than on the CRT values computation, but I won't go into detail on that here.

Shared Private Exponents or Moduli

Now I'll show you why your public key shouldn't have the same modulus n as that of someone else.

Private keys belonging to different systems or people should have different private exponents, d, even if the keys use different moduli. Or you could try your own value of d to decrypt messages encrypted for other entities, until you hit one that shares the same d. By the same token, different key pairs should have different n modulus values, even if they have different ds, because p and q are usually part of the private key. Hence, if I share the same n and thus the same p and q, I can compute your private key from your public key e using p and q.

Imagine that you know your private exponent d_1 and the public exponent e_2 of another person with whom you share the same modulus n, but without knowing its factors p and q. How can you calculate p and q from your private exponent d_1 to find the private exponent d_2 of the other person? The solution is a bit technical but elegant.

Remember that d and e satisfy $ed = k\varphi(n) + 1$, where $\varphi(n)$ is secret and could reveal p and q. You don't know k or $\varphi(n)$, but you can compute $k\varphi(n) = ed - 1$.

What can you do with $k\varphi(n)$? According to *Euler's theorem*, for any number a coprime with n, you have $a^{\varphi(n)} = 1 \bmod n$. Therefore, modulo n you have the following:

$$a^{k\varphi(n)} = \left(a^{\varphi(n)}\right)^k = 1^k = 1 \bmod n$$

Because $k\varphi(n)$ is an even number, you can write it as $2^s t$ for some numbers s and t. That is, you'll be able to write $a^{k\varphi(n)} = 1 \bmod n$ under the form $x^2 = 1 \bmod n$ for some x easily computed from $k\varphi(n)$. You can call such an x a *root of unity*.

The key observation is that $x^2 = 1 \bmod n$ is equivalent to saying that the value $x^2 - 1 = (x - 1)(x + 1)$ divides n. In other words, $x - 1$ or $x + 1$ must have a common factor with n, which can give you the factorization of n.

Listing 10-7 shows a Python implementation of this method where, to find the factors p and q from n and d, I use small, 64-bit numbers for the sake of simplicity.

```
from math import gcd

n = 36567232109354321
e = 13771927877214701
d = 15417970063428857
```

```
❶ kphi = d*e - 1
  t = kphi

❷ while t % 2 == 0:
      t = divmod(t, 2)[0]

❸ a = 2
  while a < 100:
    ❹ k = t
      while k < kphi:
          x = pow(a, k, n)
        ❺ if x != 1 and x != (n - 1) and pow(x, 2, n) == 1:
            ❻ p = gcd(x - 1, n)
              break
          k = k*2
      a = a + 2

  q = n//p
❼ assert (p*q) == n
  print('p = ', p)
  print('q = ', q)
```

Listing 10-7: A Python program that computes the prime factors p *and* q *from the private exponent* d

This program determines $k\varphi(n)$ from e and d ❶ by finding the number t such that $k\varphi(n) = 2^s t$, for some s ❷. Then it looks for a and k such that $(a^k)^2 = 1 \bmod n$ ❸, using t as a starting point for k ❹. When this condition is satisfied ❺, you've found a solution. It then determines the factor p ❻ and verifies ❼ that the value of pq equals the value of n. It then prints the resulting values of p and q:

```
p = 2046223079
q = 17870599
```

The program correctly returns the two factors.

Further Reading

RSA deserves a book by itself. I had to omit many important and interesting topics, such as Daniel Bleichenbacher's padding oracle attack on OAEP's predecessor (the standard PKCS#1 v1.5) or Manger's padding oracle attack on OAEP, both attacks being similar in spirit to the padding oracle attack on block ciphers in Chapter 4. There's also Michael Wiener's attack on RSA with low private exponents, and attacks using Coppersmith's method on RSA with small exponents that potentially also have insecure padding.

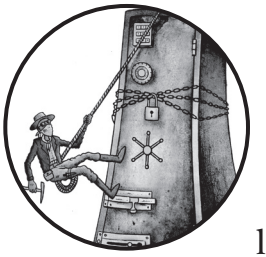
To see research results related to side-channel attacks and defenses, view the CHES workshop proceedings that have run since 1999 at *https://ches.iacr.org*. One of the most useful references while writing this chapter was

Dan Boneh's "Twenty Years of Attacks on the RSA Cryptosystem," a survey that reviews and explains the most important attacks on RSA. For reference on timing attacks, the paper "Remote Timing Attacks Are Practical" by Billy Bob Brumley and Dan Boneh is a must-read, both for its analytical and experimental contributions. To learn more about fault attacks, read the full version of the Bellcore attack paper "On the Importance of Eliminating Errors in Cryptographic Computations" by Dan Boneh, Richard DeMillo, and Richard Lipton.

The best way to learn how RSA implementations work, though sometimes painful and frustrating, is to review the source code of widely used implementations. For example, see RSA and its underlying big-number arithmetic implementations in OpenSSL, NSS (the library used by the Mozilla Firefox browser), Crypto++, or other popular software, and examine their implementations of arithmetic operations as well as their defenses against timing and fault attacks.

11

DIFFIE–HELLMAN



In November 1976, Stanford researchers Whitfield Diffie and Martin Hellman published a research paper titled "New Directions in Cryptography" that revolutionized cryptography forever. Their paper introduced the notion of public-key encryption and signatures, though they didn't actually have any of those schemes; they simply had what they termed a *public-key cryptosystem*, a protocol that allows two parties to establish a shared secret by exchanging information visible to an eavesdropper. This is now known as the *Diffie–Hellman (DH) protocol*. Prior to Diffie–Hellman, establishing a shared secret required tedious procedures such as manually exchanging sealed envelopes.

Once communicating parties establish a shared secret value with the DH protocol, they can use that secret to establish a *secure channel* by turning the secret into one or more symmetric keys that they then use to encrypt and authenticate subsequent communication. The DH protocol and its variants are therefore called *key agreement* protocols.

In the first part of this chapter, you'll read about the mathematical foundations of the Diffie–Hellman protocol, including the computational problems that DH relies on to perform its magic. Then you'll learn about different versions of the Diffie–Hellman protocol you can use to create secure channels. Finally, because Diffie–Hellman schemes are secure only when their parameters are well chosen, you'll see scenarios where Diffie–Hellman can fail.

NOTE *Diffie and Hellman received the prestigious Turing Award in 2015 for their invention of public-key cryptography and digital signatures, but others deserve credit as well. In 1974, while a computer science undergraduate, Ralph Merkle introduced the idea of public-key cryptography with* Merkle's puzzles. *Around that same time, researchers at the British Government Communications Headquarters (GCHQ), the British equivalent of the NSA, discovered the principles behind Rivest–Shamir–Adleman (RSA) and Diffie–Hellman key agreement, though that fact was declassified only decades later.*

The Diffie–Hellman Function

To understand DH key agreement protocols, you must understand their core operation, the *DH function*. Diffie and Hellman originally defined the DH function to work with groups denoted $\mathbf{Z}_p^*$, which consist of nonzero integer numbers modulo a prime number, which is usually denoted p (see Chapter 9). Another public parameter is the base number, or *generator*, g.

The DH function involves two private values chosen randomly by the two communicating parties from the group $\mathbf{Z}_p^*$, which we'll write a and b. A private value a is associated with the public value $A = g^a \bmod p$, or g raised to the power a modulo p. This A is sent to the other party through a message visible to eavesdroppers. The public value associated with b is $B = g^b \bmod p$, which is sent to the owner of a. An attacker can thus learn A and B.

DH works by combining either public value with the other private value, such that the result is the same in both cases: $A^b = (g^a)^b = g^{ab}$ and $B^a = (g^b)^a = g^{ba} = g^{ab}$. The resulting value, g^{ab}, is the *shared secret*; you then pass it to a *key derivation function (KDF)* to generate one or more shared symmetric keys. A KDF is a kind of hash function that returns a random-looking string the size of the desired key length.

And that's it. Like many great scientific discoveries (gravity, relativity, quantum computing, or RSA), the Diffie–Hellman trick is relatively simple in hindsight.

Diffie–Hellman's simplicity can be deceiving, however. For one, it won't work with just any prime p or base number g. Some values of g restrict the shared secrets g^{ab} to a small subset of possible values, whereas you'd expect

to have about as many possible values as elements in $\mathbf{Z}_p^*$ and therefore as many possible values for the shared secret. To ensure the highest security, safe DH parameters should work with a prime p such that $(p - 1)/2$ is also prime. Such a *safe prime* guarantees that the group doesn't have small subgroups that would make DH easier to break. With a safe prime, DH can work with any element in $\mathbf{Z}_p^*$, excepting 1 and $p - 1$; notably, $g = 2$ makes computations slightly faster. But generating a safe prime p takes more time than generating a totally random one.

For example, the `dhparam` command of the OpenSSL toolkit generates only safe DH parameters, but the extra checks built into the algorithm result increase the execution time considerably, as Listing 11-1 shows.

```
$ time openssl dhparam 2048
Generating DH parameters, 2048 bit long safe prime, generator 2
This is going to take a long time
--snip--
-----BEGIN DH PARAMETERS-----
MIIBCAKCAQEAoSIbyA9e844q7V89rcoEV8vd/l2svwhIIjG9EPwWWr7FkfYhYkU9
fRNttmilGCTfxc9EDf+4dzw+AbRBc6oOL9gxUoPnOd1/G/YDYgyplF5M3xeswqea
SD+B7628pWTaCZGKZham7vmiN8azGeaYAucckTkjVWceHVIVXe5fvU74k7+C2wKk
iiyMFm8th2zm9W/shiKNV2+SsHtD6r3ZC2/hfu7XdOI4iT6ise83YicU/cRaDmK6
zgBKn3SlCjwL4M3+m1J+VhOUFz/nWTJ1IWAVC+aoLK8upqRgApOgHkVqzP/CgwBw
XAOE8ncQqroJomUSB5eLqfpAvyBWpkrwQwIBAg==
-----END DH PARAMETERS-----
openssl dhparam 2048  7.46s user 0.10s system 99% cpu 7.593 total
```

Listing 11-1: Measuring the execution time of generating 2,048-bit Diffie–Hellman parameters with the OpenSSL toolkit

It took around eight seconds to generate the DH parameters using the OpenSSL toolkit (it's common to observe generation times in the order of 30 seconds or even more than 1 minute).

For the sake of comparison, Listing 11-2 shows how long it takes on the same system to generate RSA parameters of the same size (that is, two prime numbers, p and q, each half the size of the p used for DH).

```
$ time openssl genrsa 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
.....................................................+++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
--snip--
-----END RSA PRIVATE KEY-----
openssl genrsa 2048  0.16s user 0.01s system 95% cpu 0.171 total
```

Listing 11-2: Generating 2,048-bit RSA parameters while measuring the execution time

Generating DH parameters took about 50 times longer than generating RSA parameters of the same security level, mainly due to the extra constraint imposed on the prime generated to create DH parameters.

The Diffie–Hellman Problems

The security of DH protocols relies on the hardness of computational problems, especially on that of the discrete logarithm problem (DLP) from Chapter 9. You can break DH by recovering the private value a from its public value g^a, which boils down to solving a DLP instance. But we don't care about just the discrete logarithm problem when using DH to compute shared secrets. We also care about two DH-specific problems.

The Computational Problem

The *computational Diffie–Hellman (CDH)* problem is that of computing the shared secret g^{ab} given only the public values g^a and g^b, without knowing the secret values a and b. The motivation is to ensure that even if an eavesdropper captures g^a and g^b, they shouldn't be able to determine the shared secret g^{ab}.

If you can solve DLP, then you can also solve CDH; that is, if you determine a and b given g^a and g^b, then you'll be able to compute g^{ab}. In other words, DLP is *at least* as hard as CDH. But you don't know for sure whether CDH is at least as hard as DLP, which would make the problems equally hard. In other words, DLP is to CDH what the factoring problem is to the RSA problem. (Recall that factoring allows you to solve the RSA problem but not necessarily the converse.)

Diffie–Hellman shares another similarity with RSA in that DH delivers a similar security level as RSA for a given modulus size. For example, the DH protocol with a 2,048-bit prime p offers roughly 90-bit security, as RSA with a 2,048-bit modulus. Indeed, the fastest way to break CDH is to solve DLP using the *number field sieve* algorithm, a method similar but not identical to the general number field sieve (GNFS), which breaks RSA by factoring its modulus.

The Decisional Problem

When you need a seemingly stronger assumption than CDH's hardness, enter the *decisional Diffie–Hellman (DDH)* problem. Given g^a, g^b, and a value that's either g^{ab} or g^c for some random c (each of the two with a chance of 1/2), the DDH problem consists of determining whether g^{ab} (the shared secret corresponding to g^a and g^b) was chosen.

Relying on DDH rather than CDH is relevant in the following case: imagine that an attacker can compute only the first 32 bits of g^{ab} given the 2,048-bit values of g^a and g^b. Although CDH remains unbroken because 32 bits may not be enough to completely recover g^{ab}, the attacker would've learned something about the shared secret, which might allow them to compromise an application's security.

To ensure that an attacker can't learn anything about the shared secret g^{ab}, this value needs to be *indistinguishable* from a random group element, just as an encryption scheme is secure when ciphertexts are indistinguishable from random strings. That is, an attacker shouldn't be able

to determine whether a given number is g^{ab} or g^c for some random c, given g^a, g^b. The *decisional Diffie–Hellman assumption* assumes that no attacker can solve DDH efficiently.

If DDH is hard, then so is CDH, and you can't learn anything about g^{ab}. So if you can solve CDH, you can also solve DDH: given a triplet (g^a, g^b, x), you'd be able to derive g^{ab} from g^a and g^b and check whether the result is equal to the given x.

The bottom line is that DDH is fundamentally less hard than CDH (notably, DDH is not hard over $\mathbf{Z}_p^*$, contrarily to CDH), yet DDH hardness is a prime assumption in cryptography and one of the most studied.

Variants of Diffie–Hellman

Sometimes cryptographers devise new schemes and prove that they're at least as hard to break as it is to solve some hard problem. But such hard problems are not always CDH or DDH but instead can be variants of them. We'd like to be able to demonstrate that breaking a cryptosystem is as hard as solving CDH or DDH, but this isn't always possible with advanced cryptographic mechanisms, typically because such schemes involve more complex operations than basic Diffie–Hellman protocols.

For example, in one DH-like problem, given g^a, an attacker attempts to compute $g^{1/a}$, where $1/a$ is the inverse of a in the group (typically $\mathbf{Z}_p^*$ for some prime p). In another, an attacker might distinguish the pairs (g^a, g^b) from the pairs $(g^a, g^{1/a})$ for random a and b. Finally, in the *twin Diffie–Hellman problem*, given g^a, g^b, and g^c, an attacker attempts to compute the two values g^{ab} and g^{ac}. Sometimes such DH variants turn out to be as hard as CDH or DDH, and sometimes they're fundamentally easier—and therefore provide lower security guarantees. As an exercise, try to find connections between the hardness of these problems and that of CDH and DDH. (Twin Diffie–Hellman is actually *as hard* as CDH, but that isn't easy to prove!)

Key Agreement Protocols

The Diffie–Hellman problem is designed to build secure key agreement protocols, which secure communication between two or more parties communicating over a network with the aid of a shared secret. The parties turn this secret into one or more *session keys*—symmetric keys that encrypt and authenticate the information exchanged for the duration of the session.

Before studying actual DH protocols, you should know what makes a key agreement protocol secure and how simpler protocols work. We'll begin our discussion with a prevalent key agreement protocol that doesn't rely on DH.

Non-DH Key Agreement

To provide a sense of how a key agreement protocol works and what it means for it to be secure, let's look at the protocol the 4G and 5G

telecommunications standards use to establish communication between a SIM card and a telecom operator: *authenticated key agreement (AKA)*. It doesn't use the Diffie–Hellman function but instead uses only symmetric-key operations. Figure 11-1 details how the protocol works.

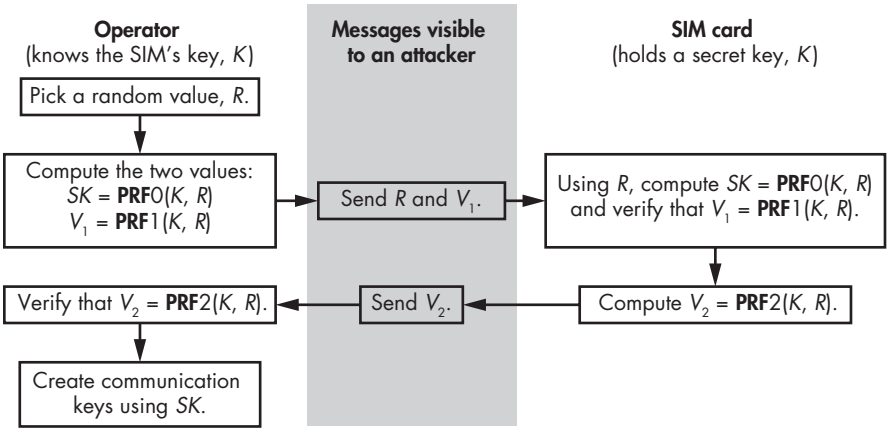

Operator
(knows the SIM's key, K)
Messages visible to an attacker
SIM card
(holds a secret key, K)
Pick a random value, R.
Compute the two values:
SK = PRF0(K, R)
V1 = PRF1(K, R)
Send R and V1.
Using R, compute SK = PRF0(K, R)
and verify that V1 = PRF1(K, R).
Verify that V2 = PRF2(K, R).
Send V2.
Compute V2 = PRF2(K, R).
Create communication keys using SK.

Figure 11-1: The AKA protocol in 4G and 5G telecommunication

In this description of the protocol, the SIM card has a secret key, K, that the operator knows. The operator begins the session by selecting a random value, R, and then computes two values, SK and V_1, based on two pseudo-random functions, **PRF**0 and **PRF**1. Next, the operator sends a message to the SIM card containing the values R and V_1, which are visible to attackers. Once the SIM card has R, it has what it needs to compute SK with **PRF**0, and it does. The two parties in this session end up with a shared key, SK, that attackers are unable to determine by simply looking at the messages exchanged between the parties, or even by modifying them or injecting new ones. The SIM card verifies that it's talking to the operator by recomputing V_1 with **PRF**1, K, and R, and then checking to make sure that the calculated V_1 matches the V_1 sent by the operator. The SIM card then computes a verification value, V_2, with a new function, **PRF**2, with K and R as input, and sends V_2 to the operator. The operator verifies that the SIM card knows K by computing V_2 and checking that the computed value matches the received V_2.

In the protocol as I described it, there's a way to fool the SIM card with a replay attack. Essentially, if an attacker captures a pair (R, V_1), they may send it to the SIM card and trick the SIM into believing that the pair came from a legitimate operator that knows K. To prevent this attack, the protocol includes additional checks to ensure the same R isn't reused.

Problems arise if K is compromised. For example, an attacker who compromises K can perform a man-in-the-middle attack and listen to all cleartext communication. Such an attacker could send messages between the two parties while pretending to be both the legitimate SIM card operator and the SIM card. Even if K isn't compromised at the time of a given

communication, an attacker can record communications and any messages exchanged during the key agreement and later decrypt those communications by using the captured R values if they find K. An attacker could then determine the past session keys and use them to decrypt the recorded traffic—in that case, the protocol doesn't offer *forward secrecy*.

Attack Models for Key Agreement Protocols

There is no single definition of security for key agreement protocols, and a key protocol is never completely secure without context and without considering the attack model and the security goals. You can, for example, argue that the previous 4G/5G protocol is secure because a passive attacker won't find the session keys, but it's also insecure because once the key K leaks, this compromises all previous and future communications.

There are different notions of security in key agreement protocols as well as three main attack models that depend on the information the protocol leaks. From weakest to strongest, these are the *network attacker*, the *data leak*, and the *breach*:

The network attacker This attacker observes the messages exchanged between the two legitimate parties running a key agreement protocol and can record, modify, drop, or inject messages. To protect against such an attacker, a key agreement protocol must not leak any information on the established shared secret.

The data leak In this model, the attacker acquires the session key and all *temporary* secrets (such as SK in the telecom protocol example) from one or more executions of the protocol, but not the long-term secrets (like K in that same protocol).

The breach (or corruption) In this model, the attacker learns the long-term key of one or more of the parties. Once a breach occurs, security is no longer attainable because the attacker can impersonate one or both parties in subsequent sessions of the protocol, as it's the only piece of information that identifies a party (at least in theory, since in practice mechanisms such as IP whitelisting can reduce the risk of impersonation). Nonetheless, the attacker shouldn't be able to recover secrets from sessions executed before gathering the key.

Now that we've looked at the attack models and seen what an attacker can do, let's explore the security goals—that is, the security guarantees that the protocol should offer. You can design a key agreement protocol to satisfy several security goals. The four most relevant ones are described here, from simplest to most sophisticated:

Authentication The protocol should allow for *mutual authentication*, wherein each party can authenticate the other party. AKA occurs when a protocol authenticates both parties.

Key control Neither party should be able to choose the final shared secret or coerce it to be in a specific subset. The previously discussed

4G/5G key agreement protocol lacks this property because the operator chooses the value for R that entirely determines the final shared key.

Forward secrecy Even if all long-term secrets are exposed, an attacker should be unable to compute shared secrets from previous executions of the protocol, even if they record all previous executions or can inject or modify messages from previous executions. A *forward-secret*, or *forward-secure*, protocol guarantees that even if you have to deliver your devices and their secrets to some authority, they won't be able to decrypt your prior encrypted communications. (The 4G/5G key agreement protocol doesn't provide forward secrecy.)

Resistance to key-compromise impersonation (KCI) KCI occurs when an attacker compromises a party's long-term key and can use it to impersonate another party. For example, the 4G/5G key agreement protocol allows trivial key-compromise impersonation because both parties share the same key K. A key agreement protocol ideally prevents this kind of attack.

Performance

To be useful, a key agreement protocol should be efficient as well as secure. You should take several factors into account when considering a key agreement protocol's efficiency, including the number of messages exchanged, their length, the computational effort to implement the protocol, and whether precomputations can be made to save time. A protocol is generally more efficient when exchanging fewer, shorter messages, and it's best if interactivity is kept minimal so that neither party has to wait to receive a message before sending the next one. You can typically measure a protocol's efficiency through its duration in terms of *round trips*, or the time it takes to send a message and receive a response.

Round-trip time is usually the main cause of latency in protocols, but the amount of computation to be carried out by the parties also counts; the fewer required computations, and the more precomputations that can be done in advance, the better. For example, the 4G/5G key agreement protocol exchanges two messages of a few hundred bits each, which must be sent in a certain order. You can use precomputation with this protocol to save time since the operator can pick many values of R in advance; precompute the matching values of SK, V_1, and V_2; and store them all in a database. In this case, precomputation has the advantage of reducing the exposure of the long-term key.

Diffie–Hellman Protocols

The Diffie–Hellman function is the core of most of the deployed public-key agreement protocols—for example, in TLS and SSH. However, there is no single Diffie–Hellman protocol but rather a variety of ways to use the DH function to establish a shared secret. We'll review three protocols in the sections that follow. In each discussion, I'll stick to the usual crypto placeholder names and call the two parties Alice and Bob, and the attacker Eve. I'll write g as the generator of the group used for arithmetic operations, a value fixed and known in advance to Alice, Bob, and Eve.

Anonymous Diffie–Hellman

Anonymous Diffie–Hellman is the simplest Diffie–Hellman protocol. It's anonymous because it's not authenticated; the participants have no cryptographic identity that either party can verify, and neither party holds a long-term key. Alice can't prove to Bob that she's Alice, and vice versa.

In anonymous Diffie–Hellman, each party picks a random value (a for Alice and b for Bob) to use as a private key and sends the corresponding public key to the other peer. Figure 11-2 shows the process in more detail.

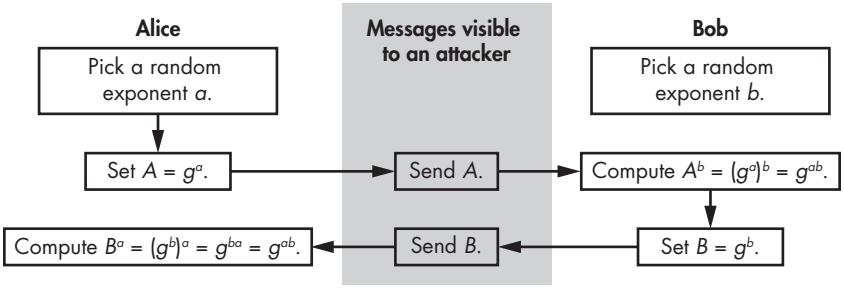


Figure 11-2: The anonymous Diffie–Hellman protocol

Alice uses her exponent a and the group basis g to compute $A = g^a$, which she sends to Bob. Bob receives A and computes A^b, which is equal to $(g^a)^b$. Bob now obtains the value g^{ab} and computes B from his random exponent b and the value g. He then sends B to Alice, which she uses to compute g^{ab}. Alice and Bob end up with the same value, g^{ab}, after performing similar operations that involve raising both g and the value received to their private exponent's power. A simple protocol, secure against only the laziest of attackers.

Attackers can take down anonymous DH with a man-in-the-middle attack. A network attacker simply needs to intercept messages and pretend to be Bob (to Alice) and pretend to be Alice (to Bob), as Figure 11-3 depicts.

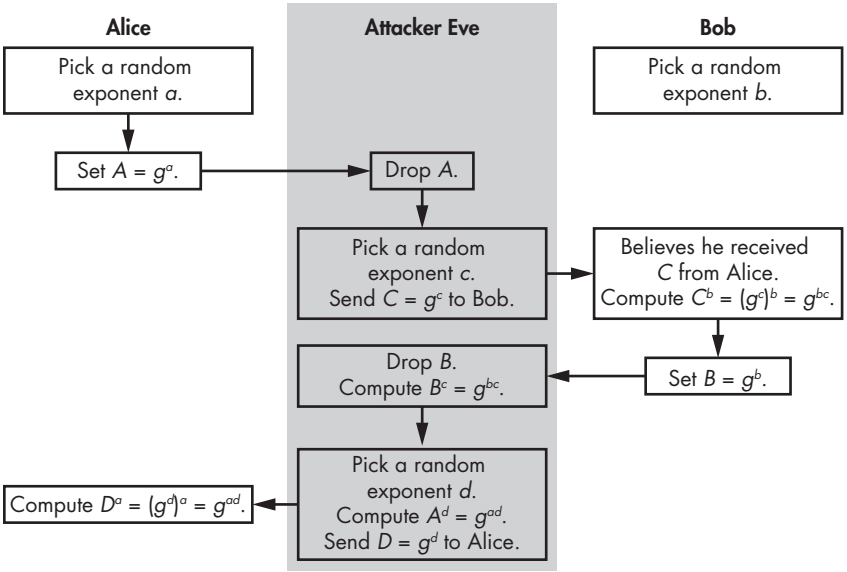

Alice
Attacker Eve
Bob
Pick a random exponent a.
Pick a random exponent b.
Set A = g^a.
Drop A.
Pick a random exponent c.
Send C = g^c to Bob.
Believes he received C from Alice.
Compute C^b = (g^c)^b = g^bc.
Drop B.
Compute B^c = g^bc.
Set B = g^b.
Pick a random exponent d.
Compute A^d = g^ad.
Send D = g^d to Alice.
Compute D^a = (g^d)^a = g^ad.

Figure 11-3: A man-in-the-middle attack on the anonymous Diffie–Hellman protocol

As in the previous exchange, Alice and Bob pick random exponents, a and b. Alice now computes and sends A, but Eve intercepts and drops the message. Eve then picks a random exponent, c, and computes $C = g^c$ to send to Bob. Because this protocol has no authentication, Bob believes he is receiving C from Alice and goes on to compute g^{bc}. Bob then computes B and sends that value to Alice, but Eve intercepts and drops the message again. Eve now computes g^{bc}; picks a new exponent, d; computes g^{ad}; computes D from g^d; and sends D to Alice. Alice then computes g^{ad} as well.

As a result of this attack, the attacker Eve shares a secret with Alice (g^{ad}) and another secret with Bob (g^{bc}), while Alice and Bob believe that they're sharing a single secret with each other. After completing the protocol execution, Alice derives symmetric keys from g^{ad} to encrypt data sent to Bob, but Eve intercepts the encrypted messages, decrypts them, and reencrypts them to Bob using another set of keys derived from g^{bc}—after potentially modifying the cleartext. All of this happens with Alice and Bob unaware; they're doomed.

To foil this attack, you need a way to authenticate the parties so Alice can prove she's the real Alice and Bob can prove he's the real Bob. Fortunately, there's a way to do so.

Authenticated Diffie–Hellman

Authenticated Diffie–Hellman addresses the man-in-the-middle attacks that can affect anonymous DH. Authenticated DH equips the two parties with both a private key and a public key, thereby allowing Alice and Bob to sign their messages to stop Eve from sending messages on their behalf. Here, the signatures aren't computed with a DH function but with a public-key signature scheme such as RSA-PSS. As a result, to successfully send messages on behalf

of Alice, an attacker needs to forge a valid signature, which is impossible with a secure signature scheme. Figure 11-4 shows how authenticated DH works.

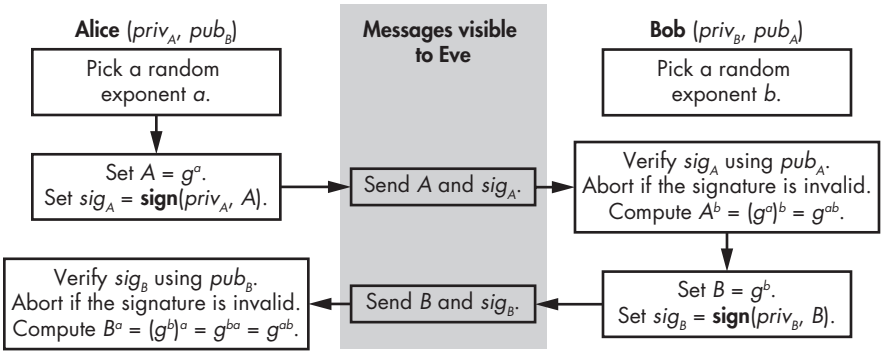

Alice ($priv_A$, pub_B)
Pick a random exponent a.
Set $A = g^a$.
Set sig_A = **sign**($priv_A$, A).
Messages visible to Eve
Send A and sig_A.
Send B and sig_B.
Bob ($priv_B$, pub_A)
Pick a random exponent b.
Verify sig_A using pub_A.
Abort if the signature is invalid.
Compute $A^b = (g^a)^b = g^{ab}$.
Set $B = g^b$.
Set sig_B = **sign**($priv_B$, B).
Verify sig_B using pub_B.
Abort if the signature is invalid.
Compute $B^a = (g^b)^a = g^{ba} = g^{ab}$.

Figure 11-4: The authenticated Diffie–Hellman protocol

The label **Alice** ($priv_A$, pub_B) on the first line means that Alice holds her own private key, $priv_A$, as well as Bob's public key, pub_B. This *priv/pub* key pair is called a *long-term key* because it's fixed in advance and remains constant through consecutive runs of the protocol. Alice can use her key pair $priv_A/pub_A$ with parties other than Bob, as long as they know pub_A (*how* they know it is another question and one of the hardest operational problems in cryptography). These long-term private keys should be kept secret, while the public keys are considered to be known to an attacker.

Alice and Bob begin by picking random exponents, a and b, as in anonymous DH. Alice then calculates A and a signature sig_A based on a combination of her signing function **sign**, her private key $priv_A$, and A. Now Alice sends A and sig_A to Bob, who verifies sig_A with her public key pub_A. If the signature is invalid, Bob knows that the message didn't come from Alice, and he discards A.

If the signature is correct, Bob computes g^{ab} from A and his random exponent b. He then computes B and his own signature from a combination of the **sign** function, his private key $priv_B$, and B. He sends B and sig_B to Alice, who attempts to verify sig_B with Bob's public key pub_B. Alice computes g^{ab} only if Bob's signature is successfully verified.

Security Against Network Attackers

Authenticated DH is secure against network attackers because they can't learn any bit of information on the shared secret g^{ab} since they ignore the DH exponents. Authenticated DH also provides forward secrecy: even if an attacker corrupts any of the parties at some point, as in the *breach* attack model discussed earlier, they'd learn the private signing keys but not any of the ephemeral DH exponents; hence, they'd be unable to learn the value of any previously shared secrets.

The authenticated variant of DH offers only partial protection against *key control*. Alice can't craft special values of a to restrict the choice of shared

secret g^{ab}, because she doesn't yet know g^b, which influences the result as much as a. (One exception would be if Alice chose $a = 0$, in which case you'd have $g^{ab} = 1$ for any b. The protocol should thus reject 0, though implementations may not do so in practice.) However, Bob can try several values of b until he finds one "that suits him"; for example, for which g^{ab} has certain properties, such as its first 16 bits being 1.

You can eliminate Bob's power over the value of the secret by sending **Hash**(g^b) from Bob to Alice as the first message, before Alice sends her g^a. I'll leave you to analyze this modification and understand why it works (the newly sent message is a *commitment* of Bob's public key).

Authenticated DH has other limitations. For one, Eve can pretend to be Alice by recording previous values of A and sig_A and replaying them to Bob. Bob mistakenly believes he's sharing a secret with Alice, even though Eve isn't able to learn that secret because she doesn't know Alice's secret a. She thus wouldn't be able to compute B^a from the B sent by Bob.

You can eliminate this risk by adding a *key confirmation* procedure, wherein Alice and Bob prove to each other that they own the shared secret. For example, Alice and Bob may perform key confirmation by sending **Hash**($pub_A \,||\, pub_B \,||\, g^{ab}$) and **Hash**($pub_B \,||\, pub_A \,||\, g^{ab}$), respectively, for some hash function **Hash**. Both parties can verify the correctness of these hash values by recomputing its result. The different order of public keys $pub_A \,||\, pub_B$ and $pub_B \,||\, pub_A$ ensures that Alice and Bob will send different values and that an attacker can't pretend to be Alice by copying Bob's hash value.

Security Against Data Leaks

Authenticated DH's vulnerability to data leak attackers is of greater concern. In this type of attack, the attacker learns the value of ephemeral, short-term secrets (namely, the exponents a and b) and uses that information to impersonate one of the communicating parties. If Eve learns the value of an exponent a along with the value of sig_A sent to Bob, she could initiate a new execution of the protocol and impersonate Alice, as Figure 11-5 illustrates.

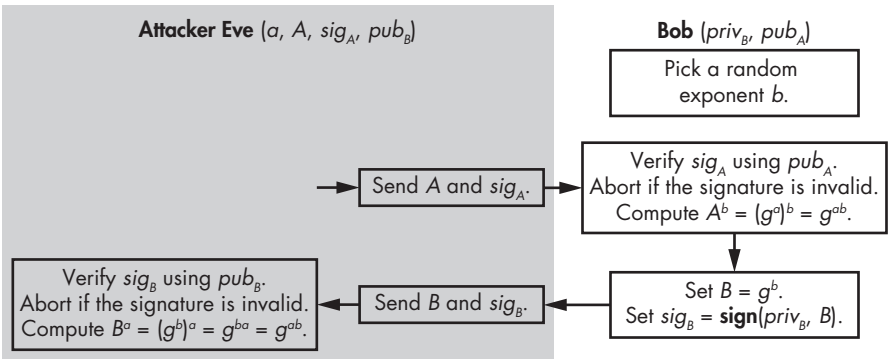

Attacker Eve (a, A, sig_A, pub_B)
Bob (priv_B, pub_A)
Pick a random exponent b.
Send A and sig_A.
Verify sig_A using pub_A.
Abort if the signature is invalid.
Compute A^b = (g^a)^b = g^ab.
Verify sig_B using pub_B.
Abort if the signature is invalid.
Compute B^a = (g^b)^a = g^ba = g^ab.
Send B and sig_B.
Set B = g^b.
Set sig_B = sign(priv_B, B).

Figure 11-5: An impersonation attack on the authenticated Diffie–Hellman protocol

In this attack scenario, Eve learns the value of an a and replays the corresponding A and its signature sig_A, pretending to be Alice. Bob verifies the signature and computes g^{ab} from A and sends B and sig_B, which Eve then uses to compute g^{ab}, using the stolen a. This results in the two having a shared secret. Bob now believes he's talking to Alice.

You can protect authenticated DH against the leak of ephemeral secrets by integrating the long-term keys into the shared secret computation so that you can't determine the shared secret without knowing the long-term secret.

Menezes–Qu–Vanstone

The *Menezes–Qu–Vanstone (MQV)* protocol is a milestone in the history of DH-based protocols. Designed in 1998, MQV was approved to protect most critical assets when the NSA included it in its Suite B, a portfolio of algorithms designed to protect classified information. (NSA eventually dropped MQV, allegedly because it wasn't used. I'll discuss the reasons why shortly.)

MQV is Diffie–Hellman on steroids. It's more secure than authenticated DH, and it improves on authenticated DH's performance properties. In particular, MQV allows users to send only two messages, independently of each other, in arbitrary order. Users can also send shorter messages than with authenticated DH, and they don't need to send explicit signature or verification messages. In other words, you don't need to use a signature scheme in addition to the Diffie–Hellman function.

As with authenticated DH, in MQV Alice and Bob each hold a long-term private key as well as the long-term public key of the other party. The difference is that the MQV keys aren't signing keys: they consist of a private exponent, x, and a public value, g^x. Figure 11-6 shows the operation of the MQV protocol.

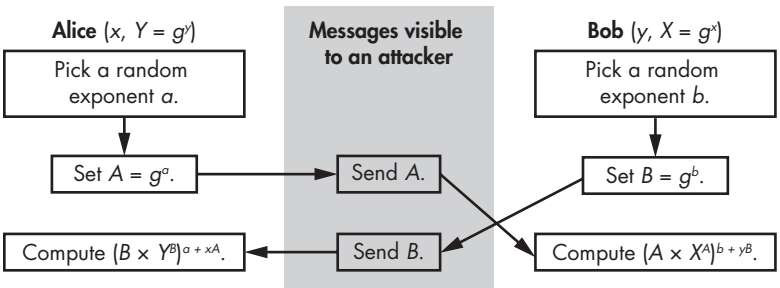

Alice (x, $Y = g^y$)
Pick a random exponent a.
Set $A = g^a$.
Compute $(B \times Y^B)^{a + xA}$.
Messages visible to an attacker
Send A.
Send B.
Bob (y, $X = g^x$)
Pick a random exponent b.
Set $B = g^b$.
Compute $(A \times X^A)^{b + yB}$.

Figure 11-6: The MQV protocol

The x and y are Alice and Bob's respective long-term private keys, and X and Y are their public keys. Bob and Alice start with their own private keys and each other's public keys, which are g to the power of a private key. Each chooses a random exponent, and then Alice calculates A and sends it to Bob, who then calculates B and sends it to Alice. Once Alice gets Bob's ephemeral public key B, she combines it with her long-term private key x,

her ephemeral private key a, and Bob's long-term public key Y by calculating the result of $(B \times Y^B)^{a + xA}$, as in Figure 11-6. Developing this expression, you obtain the following:

$$\left(B \times Y^{B}\right)^{a+xA} = \left(g^{b} \times \left(g^{y}\right)^{B}\right)^{a+xA} = \left(g^{b+yB}\right)^{a+xA} = g^{(b+yB)(a+xA)}$$

Meanwhile, Bob calculates the result of $(A \times X^A)^{b + yB}$, and you can verify that it's equal to the value Alice calculated:

$$\left(A \times X^{A}\right)^{b+yB} = \left(g^{a} \times \left(g^{x}\right)^{A}\right)^{b+yB} = \left(g^{a+xA}\right)^{b+yB} = g^{(a+xA)(b+yB)} = g^{(b+yB)(a+xA)}$$

You get the same value for both Alice and Bob, $g^{(b + yB)(a + xA)}$, which tells you that Alice and Bob share the same secret.

Unlike authenticated DH, you can't break MQV by a mere leak of the ephemeral secrets. Knowledge of a or b won't let an attacker determine the final shared secret because they need the long-term private keys to compute it.

What happens in the strongest attack model, the breach model, when a long-term key is compromised? If Eve compromises Alice's long-term private key x, the previously established shared secrets are safe because their computation also involved Alice's ephemeral private keys.

However, MQV doesn't provide *perfect* forward secrecy because of the following attack. Say, for example, that Eve intercepts Alice's A message and replaces it with her $A = g^a$ for some a that Eve chose. In the meantime, Bob sends B to Alice (and Eve records B's value) and computes the shared key. If Eve later compromises Alice's long-term private key x, she can determine the key that Bob computed during this session. This breaks forward secrecy since Eve has now recovered the shared secret of a previous execution of the protocol. In practice, however, you can eliminate the risk with a key-confirmation step in which Alice and Bob would realize that they don't share the same key, and they'd abort the protocol before deriving any session keys.

Despite its elegance and security, MQV is rarely used in practice for a couple of reasons. It used to be encumbered by patents, which hampered its widespread adoption. It's also harder than it looks to get MQV right. In fact, when weighed against its increased complexity, MQV's security benefits are often perceived as low in comparison to the simpler authenticated DH.

How Things Can Go Wrong

Diffie–Hellman protocols can fail spectacularly in a variety of ways. The following sections highlight some cases often observed in practice.

Not Hashing the Shared Secret

I've alluded to the fact that the shared secret that concludes a DH session exchange (g^{ab} in our examples) is taken as input to derive session keys but is

not a key itself. And it shouldn't be. A symmetric key should look random, and each bit should either be 0 or 1 with the same probability. But g^{ab} is not a random string; it's a random element within some mathematical group whose bits may be biased toward 0 or 1. A random group element is different from a random string of bits.

Imagine, for example, that you're working within the multiplicative group $\mathbf{Z}_{13}^{*} = \{1, 2, 3, \ldots, 12\}$ using $g = 2$ as a generator of the group, meaning that g^i spans all values of $\mathbf{Z}_{13}^{*}$ for i in 1, 2, . . . 12: $g^1 = 2$, $g^2 = 4$, $g^3 = 8$, $g^4 = 3$, and so on. If g's exponent is random, you'll get a random element of $\mathbf{Z}_{13}^{*}$, but the encoding of a $\mathbf{Z}_{13}^{*}$ element as a 4-bit string won't be uniformly random: not all bits will have the same probability of being a 0 or a 1. In $\mathbf{Z}_{13}^{*}$, seven values have 0 as their most significant bit (the numbers from 1 to 7 in the group), but only five have 1 as their most significant bit (from 8 to 12). That is, this bit is 0 with probability $7/12 \approx 0.58$, whereas a random bit should ideally be 0 with probability 0.5. Moreover, the 4-bit sequences 1101, 1110, and 1111 will never appear.

To avoid such biases in the session keys derived from a DH shared secret, use a cryptographic hash function such as BLAKE3 or SHA-3—or, better yet, a key derivation function (KDF). An example of KDF construction is HKDF, or HMAC-based KDF (as specified in RFC 5869), but today BLAKE2 and SHA-3 feature dedicated modes to behave as KDFs.

Anonymous Diffie–Hellman from TLS 1.0

The TLS protocol is the security behind HTTPS secure websites as well as many other protocols, such as email transfer with the Simple Mail Transfer Protocol (SMTP). TLS takes several parameters, including the type of Diffie–Hellman protocol it will use. For backward compatibility reasons, TLS supports anonymous DH since version 1.0 up to version 1.2 (that is, without any server authentication), though not in version 1.3. As DH is secure against only passive attackers, it can give a false impression of security.

The original documentation of TLS describes the risks of this protocol (*https://www.rfc-editor.org/rfc/rfc2246*):

> Completely anonymous connections only provide protection against passive eavesdropping. Unless an independent tamper-proof channel is used to verify that the finished messages were not replaced by an attacker, server authentication is required in environments where active man-in-the-middle attacks are a concern.

Unsafe Group Parameters

In January 2016, the maintainers of the OpenSSL toolkit fixed a high-severity vulnerability (CVE-2016-0701) that allowed an attacker to exploit unsafe Diffie–Hellman parameters. The root cause of the vulnerability was that OpenSSL allowed users to work with unsafe DH group parameters

(namely, an unsafe prime p) instead of throwing an error and aborting the protocol altogether before performing any arithmetic operation.

Essentially, OpenSSL accepted a prime number p whose multiplicative group $\mathbf{Z}_p^*$ (where all DH operations happen) contained small subgroups. As you learned at the beginning of this chapter, the existence of small subgroups within a larger group in a cryptographic protocol is bad because it confines shared secrets to a much smaller set of possible values than if it were to use the whole group $\mathbf{Z}_p^*$. Worse still, an attacker can craft a DH exponent x that, when combined with the victim's public key g^y, reveals information on the private key y and eventually its entirety.

Although the actual vulnerability is from 2016, the principle the attack used dates back to the 1997 paper "A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup" by Chae Hoon Lim and Pil Joong Lee. The fix for the vulnerability is simple: when accepting a prime p as group modulus, the protocol must check that p is a safe prime by verifying that $(p - 1) / 2$ is prime as well to ensure that the group $\mathbf{Z}_p^*$ won't have small subgroups and that an attack on this vulnerability will fail.

Further Reading

You can dig deeper into the DH key agreement protocols by reading a number of standards and official publications, including ANSI X9.42, RFC 2631 and RFC 5114, IEEE 1363, and NIST SP 800-56A. These serve as references to ensure interoperability and to provide recommendations for group parameters.

To learn more about advanced DH protocols (such as MQV and its cousins HMQV and OAKE, among others) and their security notions (including unknown-key share attacks and group representation attacks), read the 2005 article "HMQV: A High-Performance Secure Diffie–Hellman Protocol" by Hugo Krawczyk (*https://eprint.iacr.org/2005/176*) and the 2011 article "A New Family of Implicitly Authenticated Diffie–Hellman Protocols" by Andrew C. Yao and Yunlei Zhao (*https://eprint.iacr.org/2011/035*). These articles express Diffie–Hellman operations differently than in this chapter. For example, they represent the shared secret as xP instead of g^x. Generally, you'll find multiplication replaced with addition and exponentiation replaced with multiplication, because those protocols are usually not defined over groups of integers but over elliptic curves, as you'll learn in Chapter 12.

12

ELLIPTIC CURVES

The introduction of *elliptic curve cryptography (ECC)* in 1985 revolutionized public-key cryptography. It's more powerful and efficient than alternatives like RSA and classical Diffie–Hellman: ECC with a 256-bit key is stronger than RSA with a 4,096-bit key. But it's also more complex.

Like RSA, ECC consists mainly of multiplications of large numbers, but it does so to combine points on a mathematical curve, called an *elliptic curve* (this has nothing to do with an ellipse, by the way). To complicate matters, there are many types of elliptic curves—simple and sophisticated, efficient and inefficient, and secure and insecure, depending on the use case.

ECC wasn't adopted by standardization bodies until the early 2000s, and it wasn't seen in major cryptographic software until much later: OpenSSL added ECC in 2005, and the OpenSSH secure connectivity tool waited until 2011. You'll find ECC in most HTTPS connections, in mobile phones, and in blockchain platforms such as Bitcoin and Ethereum. Indeed, elliptic curves allow you to perform common public-key cryptography operations

such as encryption, signature, and key agreement faster than their classical counterparts. Most cryptographic applications that rely on the discrete logarithm problem (DLP) also work when based on its elliptic curve counterpart, ECDLP, with one notable exception: the Secure Remote Password (SRP) protocol.

This chapter focuses on applications of ECC and discusses when and why to use ECC over RSA or classical Diffie–Hellman, as well as how to choose the right elliptic curve for your application.

What Is an Elliptic Curve?

An elliptic curve is a *curve* on a plane—a set of points with x- and y-coordinates. A curve's equation defines all the points that belong to that curve. For example, the curve $y = 3$ is a horizontal line with the vertical coordinate 3, curves of the form $y = ax + b$ with fixed numbers a and b are straight lines, $x^2 + y^2 = 1$ is a circle of radius 1 centered on the origin, and so on. Whatever the type of curve, the points on a curve are (x, y) pairs that satisfy the curve's equation.

In cryptography, an elliptic curve typically has an equation in the form $y^2 = x^3 + ax + b$ (the *Weierstrass form*), where the constants a and b define the shape of the curve. For example, Figure 12-1 shows the elliptic curve that satisfies the equation $y^2 = x^3 - 4x$.

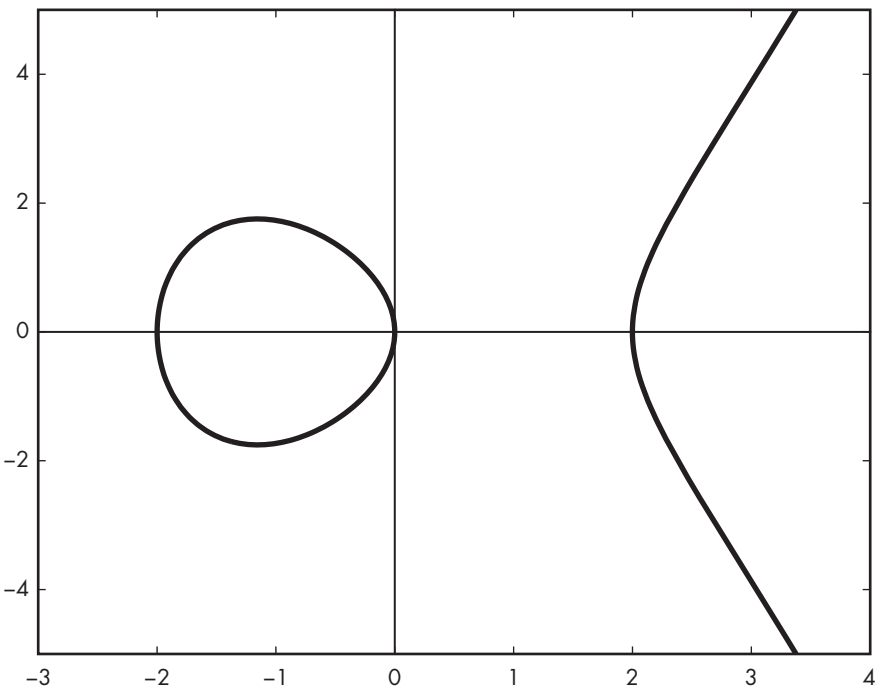


Figure 12-1: An elliptic curve with the equation $y^2 = x^3 - 4x$, shown over the real numbers

This figure displays all the points that make up the curve for x between –3 and 4, be they points on the left side of the curve, which looks like a circle, or on the right side, which looks like a parabola. All points have (x, y) coordinates that satisfy the curve's equation $y^2 = x^3 - 4x$. For example, when $x = 0$, then $y^2 = x^3 - 4x = 0^3 - 4 \times 0 = 0$; hence, $y = 0$ is a solution, and the point $(0, 0)$ belongs to the curve. Likewise, if $x = 2$, the solution to the equation is $y = 0$, meaning that the point $(2, 0)$ belongs to the curve.

NOTE *In this chapter, I focus on the simplest, most common type of elliptic curves—those with an equation like* $y^2 = x^3 + ax + b$*—but other elliptic curves have equations in different forms. For example,* Edwards curves *have equations of the form* $x^2 + y^2 = 1 + dx^2y^2$*. Cryptographers sometimes use Edwards curves (for example, in the Ed25519 scheme).*

When using elliptic curves for cryptography, it's crucial to distinguish points that belong to the curve from other points, as points off the curve often present a security risk. However, the curve's equation doesn't always admit solutions, at least not in the natural number plane. For example, to find points with the horizontal coordinate $x = 1$, you solve $y^2 = x^3 - 4x$ and obtain $y^2 = x^3 - 4x = 1^3 - 4 \times 1$, resulting in –3. But $y^2 = -3$ doesn't have a solution because there's no number for which $y^2 = -3$. There exists a solution in the complex numbers ($\sqrt{3}i$), but in practice, elliptic curve cryptography uses only natural numbers (ECC works with integer number modulo a prime number, either directly or via polynomials). Because there's no solution to the curve's equation for $x = 1$, the curve has no point at that position on the x-axis, as you can see in Figure 12-1.

If you try to solve for $x = -1$, you get the equation $y^2 = -1 + 4 = 3$, which has two solutions ($y = \sqrt{3} \approx 1.73$ and $y = -\sqrt{3} \approx -1.73$), the square root of 3 and its negative value. The square of both values is 3, and Figure 12-1 has two points with $x = -1$. More generally, the curve is symmetric with respect to the x-axis for all points that satisfy its equation (as are all elliptic curves of the form $y^2 = x^3 + ax + b$, except for $y = 0$).

Elliptic Curves over Integers

Here's a bit of a twist: the curves in elliptic curve cryptography don't actually look like Figure 12-1. They're neither curves nor ellipses. They look instead like Figure 12-2, which is a cloud of points rather than a curve. What's going on here?

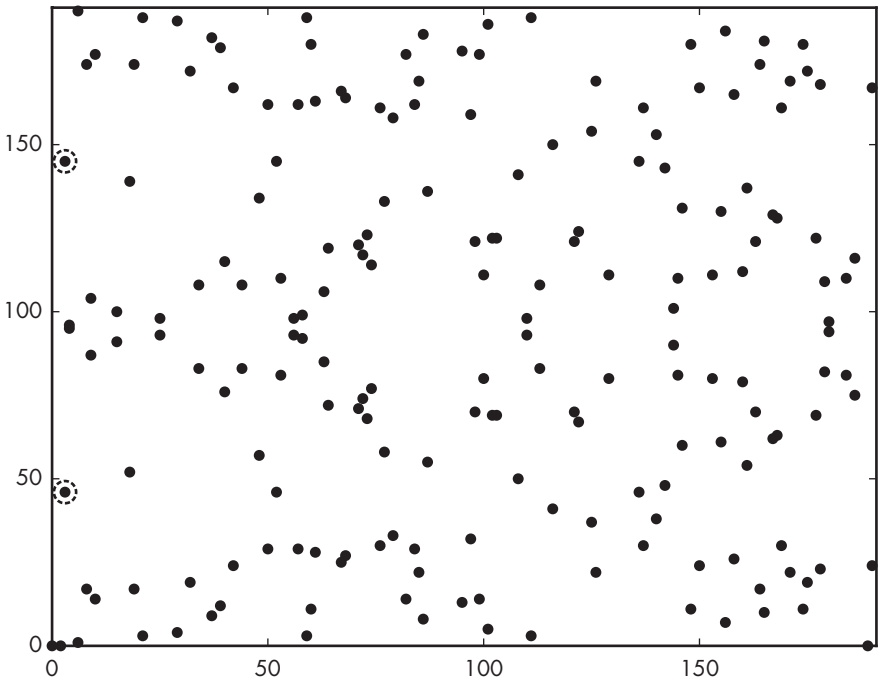


Figure 12-2: The elliptic curve with the equation $y^2 = x^3 - 4x$ over Z_{191}, the set of integers modulo 191

Figures 12-1 and 12-2 are based on the same curve equation, $y^2 = x^3 - 4x$, but they show the curve's points with respect to different sets of numbers: Figure 12-1 shows the curve's points over the set of *real numbers*, which includes negative numbers, decimals, and so on. For example, as a continuous curve, it shows the points at $x = 2.0$, $x = 2.1$, $x = 2.00002$, and so on. Figure 12-2, on the other hand, shows only *integers* that satisfy this equation, which excludes decimal numbers. Specifically, Figure 12-2 shows the curve $y^2 = x^3 - 4x$ with respect to the integers *modulo 191*: 0, 1, 2, 3, up to 190. You can denote this set of numbers $\mathbf{Z}_{191}$. (There's nothing special with 191 here, except that it's a prime number. I picked a small number to avoid having too many points on the graph.) The points in Figure 12-2 therefore all have x- and y-coordinates that are integers modulo 191 and that satisfy the equation $y^2 = x^3 - 4x$. For example, for $x = 2$, you have $y^2 = 0$, for which $y = 0$ is a valid solution. This tells you that the point (2, 0) belongs to the curve.

If $x = 3$, you get the equation $y^2 = 27 - 12 = 15$, which admits two solutions in $\mathbf{Z}_{191}$, 46 and 145. Indeed, $46^2 = 2{,}116$, with $2{,}116 \bmod 191 = 15$, and $145^2 = 21{,}025$, with $21{,}025 \bmod 191 = 15$. Thus, the points (3, 46) and (3, 145) both belong to the curve and appear as in Figure 12-2 (the two points circled on the left).

NOTE *Figure 12-2 considers points from the set denoted* $\mathbf{Z}_{191} = \{0, 1, 2, \ldots, 190\}$*, which includes zero. This differs from the groups denoted* $\mathbf{Z}_p^*$ *(with a star superscript) that we discussed in the context of RSA and Diffie–Hellman. The reason for this difference is that you'll both multiply and add numbers, and you therefore need to ensure that the set of numbers includes addition's identity element (namely 0, such that* $x + 0 = x$ *for every* x *in* $\mathbf{Z}_{191}$*).*

Also, every number x *has an inverse with respect to addition, denoted* $-x$*, such that* $x + (-x) = 0$*. For example, the inverse of 100 in* $\mathbf{Z}_{191}$ *is 91 because 100 + 91 mod 191 = 0. Such a set of numbers, where addition and multiplication are possible and where each element* x *admits an inverse with respect to addition (denoted* $-x$*) as well as an inverse (except for the element zero) with respect to multiplication (denoted* $1 / x$*), is called a* field*. When a field has a finite number of elements, as in* $\mathbf{Z}_{191}$ *and as with all fields used for elliptic curve cryptography, it is a* finite field.

The Addition Law

Now that you know that the points on an elliptic curve are coordinates (x, y) that satisfy the curve's equation, $y^2 = x^3 + ax + b$, let's look at how to add elliptic curve points with the *addition law*.

Adding Two Points

Say you want to add two points on the elliptic curve, P and Q, to give a new point, R, that is the sum of these two points. The simplest way to understand point addition is to determine the position of $R = P + Q$ on the curve relative to P and Q based on a geometric rule: draw the line that connects P and Q, find the other point of the curve that intersects with this line, and R is the reflection of this point with respect to the x-axis. For example, in Figure 12-3, the line connecting P and Q intersects the curve at a third point between P and Q, and the point $P + Q$ is at the same x-coordinate but the inverse y-coordinate.

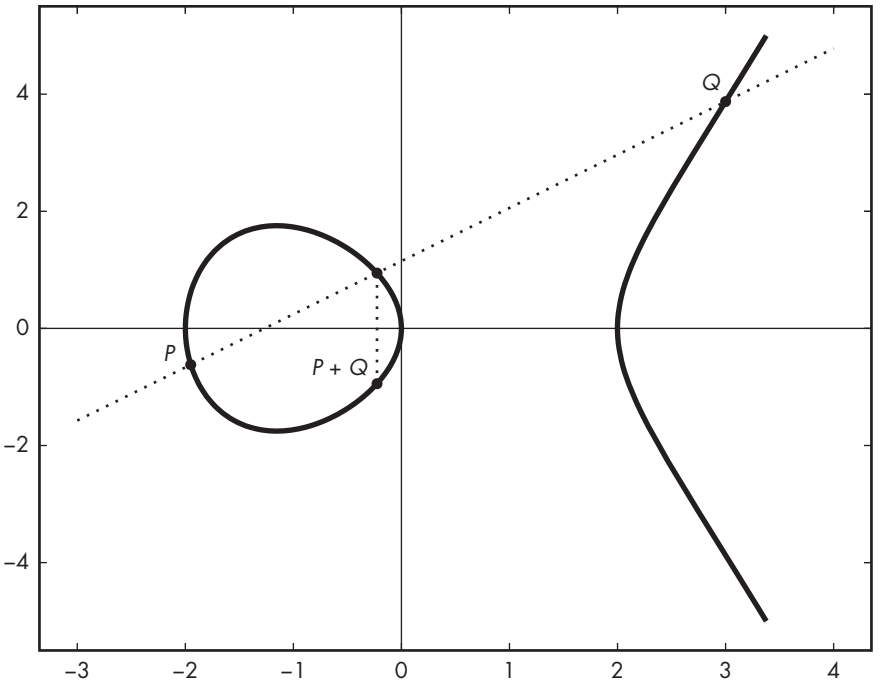


Figure 12-3: A general case of the geometric rule for adding points over an elliptic curve

This geometric rule is simple, but it won't directly give you the coordinates of the point R. You compute the coordinates (x_R, y_R) of R using the coordinates (x_P, y_P) of P and the coordinates (x_Q, y_Q) of Q using the formulas $x_R = m^2 - x_P - x_Q$ and $y_R = m(x_P - x_R) - y_P$, where the value $m = (y_Q - y_P) / (x_Q - x_P)$ is the slope of the line connecting P and Q.

Unfortunately, these formulas and the line-drawing trick in Figure 12-3 don't always work. If, for example, $P = Q$, you can't draw a line between two points (there's only one), and if $Q = -P$, the line doesn't cross the curve again, so there's no point on the curve to mirror. We'll explore these in the next sections.

Adding a Point and Its Negative

The negative of a point $P = (x_P, y_P)$ is the point $-P = (x_P, -y_P)$, which is the point mirrored around the x-axis. For any P, you can say that $P + (-P) = O$, where O is the *point at infinity*. As in Figure 12-4, the line between P and $-P$ runs to infinity and never intersects the curve. The point at infinity is to elliptic curves what zero is to integers, except that it's only a "virtual" point that you can't locate at a specific place, because it's infinitely far.

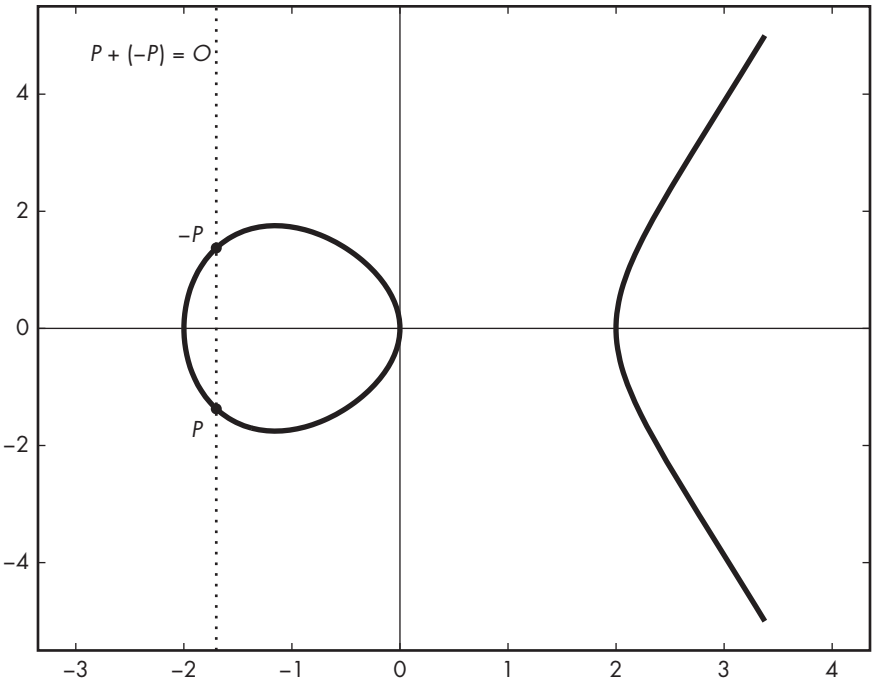


Figure 12-4: The geometric rule for adding a point and its negative, or P + (–P) = O, *when the line between the points never intersects the curve*

Doubling a Point

When $P = Q$ (that is, P and Q are at the same position), adding P and Q is equivalent to computing $P + P$, or $2P$. This addition operation is called a *doubling*.

To find the coordinates of the result $R = 2P$, you can't use the geometric rule from the previous section, because you can't draw a line between P and itself. Instead, you draw the line *tangent* to the curve at P, and $2P$ is the negation of the point where this line intersects the curve, as Figure 12-5 demonstrates.

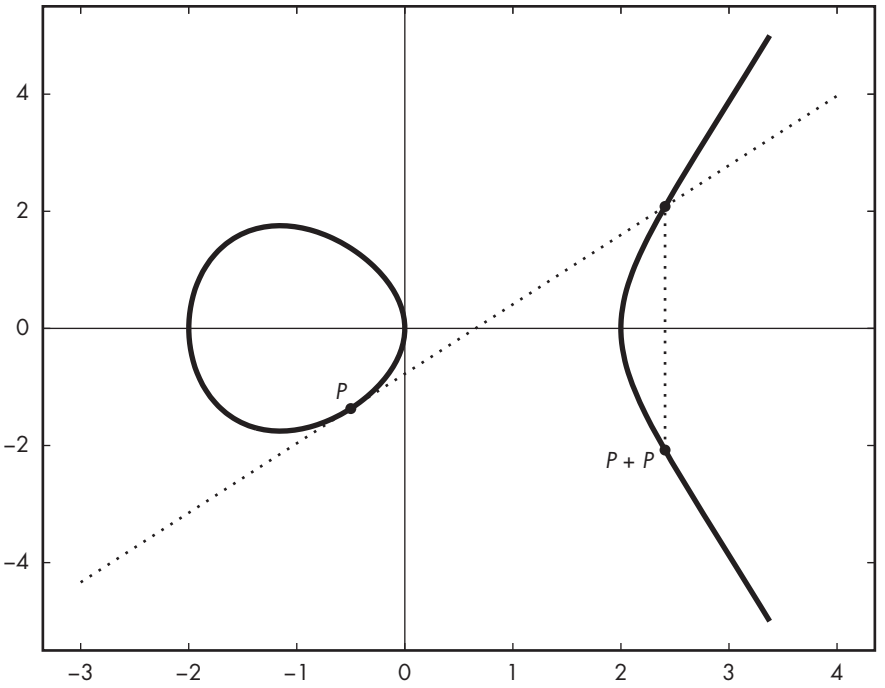


Figure 12-5: The geometric rule for doubling a point—that is, P + P

The formula to determine the coordinates (x_R, y_R) of $R = P + P$ is slightly different from the formula for a distinct P and Q. The basic formula is $x_R = m^2 - x_P - x_Q$ and $y_R = m(x_P - x_R) - y_P$, but the value of m becomes $(3x_P^2 + a) / 2y_P$, where a is the curve's parameter, as in $y^2 = x^3 + ax + b$.

Multiplying Points

To multiply points on elliptic curves by a given integer number k, you determine the point kP by adding P to itself $k - 1$ times. In other words, $2P = P + P$, $3P = P + P + P$, and so on. To obtain the x- and y-coordinates of kP, repeatedly add P to itself and apply the preceding addition law.

To compute kP efficiently, however, the naive technique of adding P by applying the addition law $k - 1$ times is far from optimal. For example, if k is large (of the order of, say, 2^{256}) as it occurs in elliptic curve–based crypto schemes, then computing $k - 1$ additions is downright infeasible.

You can gain an exponential speedup by adapting the technique in Chapter 10's "A Fast Exponentiation Algorithm" to compute $x^e \bmod n$. The difference is that you're multiplying points by an integer rather than computing exponentiations. But you can adapt the method to become *double-and-add*, where multiplication becomes addition and squaring becomes doubling.

For example, to compute $8P$ in three additions instead of seven using the naive method, first compute $P_2 = P + P$, then $P_4 = P_2 + P_2$, and finally $P_4 + P_4 = 8P$. This is the simplest case, when the multiplier is a power of two,

such as $8 = 2^3$. Otherwise, for example, to compute $10P$, proceed as follows: observing that 10 is 1010 in binary, start by the most significant ("leftmost") bit and compute the result R like so:

Bit 1 Start by setting $R = P$.

Bit 0 Double, setting $R = 2R = 2P$.

Bit 1 Double and add, setting $R = 2R + P = 2(2P) + P = 5P$.

Bit 0 Double, setting $R = 2R = 2(5P) = 10P$.

Elliptic Curve Groups

Not only can you add elliptic curve points together, you can also use them to form a group structure. According to the definition of a group (see "Groups" in Chapter 9), if the points P and Q belong to a given curve, then $P + Q$ also belongs to the curve.

Furthermore, because addition is *associative*, you have $(P + Q) + R = P + (Q + R)$ for any points P, Q, and R. In a group of elliptic curve points, we call the identity element the point at infinity and denote it as O, such that $P + O = P$ for any P. Every point $P = (x_P, y_P)$ has an inverse, $-P = (x_P, -y_P)$, such that $P + (-P) = O$.

In practice, most elliptic curve–based cryptosystems work with x- and y-coordinates that are numbers modulo a prime number, p (in other words, numbers in the finite field $\mathbf{Z}_p$). Just as the security of RSA depends on the size of the numbers used, the security of an elliptic curve–based cryptosystem depends on the number of points on the curve. But how do you know the number of points on an elliptic curve, or its *cardinality*? Well, it depends on the curve and the value of p.

NOTE *Prime fields aren't the only fields used in elliptic curve cryptography. There are also* binary fields, *which are extensions of the two-element field (which includes 0 and 1), whose elements are represented as polynomials with binary coefficients. Arithmetic over binary fields has the advantage of often being easier to implement efficiently in hardware as logic circuits, compared to a general-purpose microprocessor.*

There are of the order of p points on a curve defined over $\mathbf{Z}_p$, but you can compute the exact number of points with Schoof's algorithm, which counts points on elliptic curves over finite fields. You'll find this algorithm built into SageMath. For example, Listing 12-1 uses Schoof's algorithm to count the number of points on the curve $y^2 = x^3 - 4x$ over $\mathbf{Z}_{191}$ from Figure 12-1.

```
sage: Z = Zmod(191)
sage: E = EllipticCurve(Z, (-4,0))
sage: E.cardinality()
192
```

Listing 12-1: Computing the cardinality, or the number of points on a curve

In Listing 12-1, you first define the variable `Z` as the set over integers modulo 191; then you define the variable `E` as the elliptic curve over `Z` with the coefficients –4 and 0. Finally, you compute the number of points on the curve, or its *cardinality*, *group order*, or just *order*. This count includes the point at infinity O.

The ECDLP Problem

Chapter 9 introduced the discrete logarithm problem (DLP): that of finding the number y given some base number g, where $x = g^y \bmod p$ for some large prime number p. Cryptography with elliptic curves has a similar problem of finding the number k given a base point P where the point $Q = kP$. This is the elliptic curve discrete logarithm problem. Instead of numbers, the elliptic curve's problems operate on points and use multiplication instead of exponentiation.

About all elliptic curve cryptography is built on the ECDLP problem, which, like DLP, is believed to be hard and has withstood cryptanalysis since its introduction in 1985. One important difference from the classical DLP is that ECDLP allows you to work with smaller numbers and still enjoy a similar level of security.

Generally, when the parameter p is n-bit, ECC gets you a security level of about $n/2$ bits. For example, an elliptic curve taken over numbers modulo a 256-bit p gives a security level of about 128 bits. For the sake of comparison, to achieve a similar security level with DLP or RSA, you need to use numbers of several thousands of bits. Using smaller numbers for ECC arithmetic is one reason why it's often faster than RSA or classical Diffie–Hellman.

One approach to solving ECDLP is to find a collision between two outputs, $c_1P + d_1Q$ and $c_2P + d_2Q$. The points P and Q in these equations are such that $Q = kP$ for some unknown k, and c_1, d_1, c_2, and d_2 are the numbers you need to find k.

As with the hash function in Chapter 6, a collision occurs when two different inputs produce the same output. Therefore, to solve ECDLP, you need to find points where the following is true:

$$c_1P + d_1Q = c_2P + d_2Q$$

Assuming you've found the four coefficients, you're going to recover k. For this, replace Q with the value kP, and you have the following:

$$c_1P + d_1kP = (c_1 + d_1k)P = c_2P + d_2kP = (c_2 + d_2k)P$$

This tells you that $(c_1 + d_1k)$ equals $(c_2 + d_2k)$ when taken modulo the number of points on the curve, which isn't a secret.

From this, you can deduce the following:

$$d_2k - d_1k = c_1 - c_2$$
$$k(d_2 - d_1) = c_1 - c_2$$
$$k = (c_1 - c_2) \,/\, (d_2 - d_1)$$

And you've found k, the solution to ECDLP. Of course, that's only the big picture—the details are more complex and interesting, notably regarding the way to recover the coefficients c_1, c_2, d_1, d_2.

In practice, elliptic curves extend over numbers of at least 256 bits, which makes attacking elliptic curve cryptography by finding a collision impractical because doing so takes up to 2^{128} operations (the cost of finding a collision over 256-bit numbers—see Chapter 6).

Diffie–Hellman Key Agreement over Elliptic Curves

Recall from Chapter 11 that in the classical Diffie–Hellman (DH) key agreement protocol, two parties establish a shared secret by exchanging nonsecret values. Given some fixed number g, Alice picks a secret random number a, computes $A = g^a$, and sends A to Bob. Then Bob picks a secret random b and sends $B = g^b$ to Alice. Both combine their secret key with the other's public key to produce the same $A^b = B^a = g^{ab}$.

The elliptic curve version of DH, *elliptic curve Diffie–Hellman (ECDH)*, is identical to that of classical DH but with different notations. In the case of ECC, for some fixed-point G, Alice picks a secret random number a, computes $A = aG$ (the point G multiplied by the integer a), and sends A to Bob. Bob picks a secret random b, computes the point $B = bG$, and sends it to Alice. Then both compute the same shared secret, $aB = bA = abG$.

ECDH is to the ECDLP problem what DH is to DLP: it's secure as long as ECDLP is hard. You can therefore adapt DH protocols that rely on DLP to work with elliptic curves and rely on ECDLP as a hardness assumption. For example, authenticated DH and Menezes–Qu–Vanstone (MQV) will also be secure when used with elliptic curves. In fact, MQV was first defined as working over elliptic curves.

Signing with Elliptic Curves

The main standard algorithm you can use for signing with ECC is *elliptic curve digital signature algorithm (ECDSA)*. This algorithm has replaced RSA signatures and classical DSA signatures in many applications. ECDSA is a NIST standard, is supported in the TLS and SSH protocols, and is the main signature algorithm in many blockchain platforms, including Bitcoin and Ethereum.

As with all signature schemes, ECDSA consists of a *signature generation* algorithm that the signer uses to create a signature using their private key and a *verification* algorithm that a verifier uses to check a signature's correctness given the signer's public key. The signer holds a number, d, as a private

key, and verifiers hold the public key, $P = dG$. Both know in advance what elliptic curve to use, its order (n, the number of points in the curve), and the coordinates of a base point, G.

ECDSA Signature Generation

To sign a message, the signer first hashes the message with a cryptographic hash function such as SHA-256 or BLAKE2 to generate a hash value, h, that you interpret as a number between 0 and $n - 1$. Next, the signer picks a random number, k, between 1 and $n - 1$ and computes kG, a point with the coordinates (x, y). The signer now sets $r = x \bmod n$ and computes $s = (h + rd) / k \bmod n$ and then uses these values as the signature (r, s).

The length of the signature depends on the coordinate lengths you're using. For example, when you're working with a curve where coordinates are 256-bit numbers, r and s are both 256 bits long, yielding a 512-bit-long signature.

ECDSA Signature Verification

The ECDSA verification algorithm uses a signer's public key to verify the validity of a signature.

To verify an ECDSA signature (r, s) and a message's hash, h, the verifier first computes $w = 1 / s$, the inverse of s in the signature, which is equal to $k / (h + rd) \bmod n$, since s is defined as $s = (h + rd) / k$. Next, the verifier multiplies w with h to find u according to the following formula:

$$wh = hk / (h + rd) = u$$

The verifier then multiplies w with r to find v:

$$wr = rk / (h + rd) = v$$

Given u and v, the verifier computes the point Q according to the following formula:

$$Q = uG + vP$$

Here, P is the signer's public key, which is equal to dG, and the verifier accepts the signature only if the x-coordinate of Q is equal to the value r from the signature.

This process works because, as a last step, you compute the point Q by substituting the public key P with its actual value dG:

$$uG + vdG = (u + vd)G$$

When you replace u and v with their actual values, you obtain the following:

$$u + vd = hk / (h + rd) + drk / (h + rd) = (hk + drk) / (h + rd) = k(h + dr) / (h + rd) = k$$

This tells you that $(u + vd)$ is equal to the value k, chosen during signature generation, and that $uG + vdG$ is equal to the point kG. In other words, the verification algorithm succeeds in computing point kG, the same point computed during signature generation. Validation is complete once a verifier confirms that kG's x-coordinate is equal to the r received; otherwise, the signature is rejected as invalid.

ECDSA vs. RSA Signatures

While some view elliptic curve cryptography as an alternative to RSA for public-key cryptography, ECC and RSA don't have much in common. You use RSA only for encryption and signatures, and ECC is a family of algorithms that you use to perform encryption, generate signatures, perform key agreement, and offer advanced cryptographic functionalities such as identity-based encryption (a kind of encryption that uses encryption keys derived from a personal identifier, such as an email address).

When comparing ECDSA to RSA signatures, recall that in RSA signatures, the signer uses their private key d to compute a signature as $y = x^d \bmod n$, where x is the data to be signed and y is the signature. Verification uses the public key e to confirm that $y^e \bmod n$ equals x—a process that's clearly simpler than that of ECDSA.

RSA's verification process is often faster than ECC's signature verification because it uses a small public key e. Verification of an RSA signature then consists only in an exponentiation $y^{65,537} \bmod n$.

ECC has two major advantages over RSA: shorter signatures and signing speed. Because ECC works with shorter numbers, it produces shorter signatures than RSA (hundreds of bits long, not thousands of bits), which is an obvious benefit if you have to store or transmit numerous signatures. Signing with ECDSA is also faster than signing with RSA because ECDSA's arithmetic with much smaller numbers is computationally cheaper. For example, Listing 12-2 shows that ECDSA is about 200 times faster at signing and about as fast as RSA at verifying (benchmarks on an Apple M2 processor). Note that in this example, ECDSA signatures are also shorter than RSA signatures because they're 512 bits (two elements of 256 bits each) rather than 4,096 bits.

```
$ openssl speed ecdsap256 rsa4096
                                  sign     verify     sign/s     verify/s
rsa 4096 bits                0.003297s  0.000048s      303.3      20933.4
                                  sign     verify     sign/s     verify/s
256 bit ecdsa (nistp256)     0.0000s    0.0000s    63709.1      20979.8
```

Listing 12-2: Comparing the speed of 4,096-bit RSA signatures with 256-bit ECDSA signatures

It's fair to compare the performance of these differently sized signatures because they provide a similar security level. However, in practice, many systems use RSA signatures with 2,048 bits, which is orders of magnitude less secure than 256-bit ECDSA. Thanks to its smaller modulus size,

2,048-bit RSA is faster than 256-bit ECDSA at verifying, yet still slower at signing, as Listing 12-3 shows.

```
$ openssl speed rsa2048
                              sign     verify     sign/s     verify/s
rsa 2048 bits             0.000520s  0.000013s     1923.7      77221.1
```

Listing 12-3: The speed of 2,048-bit RSA signatures

You should prefer ECDSA to RSA except when signature verification is critical *and* you don't care about signing speed, as in a sign-once, verify-many situation (for example, when a Windows executable application is signed once and then verified by all the systems executing it).

EdDSA and Ed25519

ECDSA was introduced in the early 1990s as an elliptic curve version of DSA, the digital signature standard from the NSA standardized by NIST in 1991. But in 1989, cryptographer Claus-Peter Schnorr proposed the so-called Schnorr signature scheme, an algorithm simpler and more efficient than (EC)DSA and also suitable for elliptic curves. However, Schnorr filed a patent restricting the use of his signature scheme, preventing wide adoption and standardization.

After Schnorr's patent expired in 2008, cryptographers Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang built atop Schnorr's idea to create *Edwards-curve DSA (EdDSA)*, a signature suitable only for elliptic curves, with the following advantages:

- Simpler than ECDSA
- Faster than ECDSA, both signing and verification
- Deterministic (whereas the ECDSA and Schnorr signatures used a random number for signing), eliminating the risks related to flawed randomness

Let's see how the general EdDSA works and what makes its specific instance of Ed25519 so appealing. To keep it simple, I'll provide a simplified overview of these schemes. For the full details and rationale behind these schemes, see the 2011 paper "High-Speed High-Security Signatures" and RFC 8032, "Edwards-Curve Digital Signature Algorithm (EdDSA)."

Note that I use notations similar to those in the original article, which may differ from notations previously used in this book.

EdDSA Signature

Like any signature scheme, signing with EdDSA requires a private key and a message, but unlike with ECDSA, the private key is a random byte string rather than a random (scalar) number. You derive the actual private scalar by hashing the private key string. This has several security benefits, including making it easier for applications to pick a random key by just dumping raw bytes from a PRNG, rather than ensuring that the key has the right

format and isn't weak. It also helps to efficiently derive a nonce from the key and the message, which replaces the use of a per-signature random value like in ECDSA.

Signing then works as follows, from a 256-bit private key k and a message M of arbitrary size, and given a base point B:

1. Compute $a \| h = \mathbf{Hash}(k)$, for a hash function producing 512-bit values, where the first 256 bits form the private scalar a and the last 256 bits form the string h.
2. Define the public key as $A = aB$ (in practice, this is precomputed).
3. Compute the message's "pre-hash" $r = \mathbf{Hash}(h \| M)$ and compute the elliptic curve point $R = rB$, which is the first part of the signature, the *signature point*.
4. Compute the number $S = r + \mathbf{Hash}(R, A, M) \times a$, the second part of the signature.

The signature returned is the pair (R, S).

The computation bottleneck is thus the scalar-point multiplication rB, for a fixed base point B. When messages are long, the cost of hashing the message twice may be significant too. However, unlike in ECDSA, you don't have to compute a modular inverse.

In practice, you can optimize implementations—for example, to avoid recomputing the public key for every signature. You must also set values to the right type (such as numbers subject to modular reduction) and encode them in a reliable and nonambiguous format (like curve points).

EdDSA Verification

Given a signature (R, S), a message M, a public key A, and a base point B, verification consists of checking that SB equals $R + \mathbf{Hash}(R, A, M)A$. Note that $SB = S \times B$.

If you replace S by its expression evaluated in step 4, SB is equal to the following values:

$$\left(r + \mathbf{Hash}(R, A, M) \times a\right) \times B = rB + \mathbf{Hash}(R, A, M) \times aB$$

In this expression, you know that $rB = R$ and $aB = A$. You thus end up with the expected result, $R + \mathbf{Hash}(R, A, M) \times A$.

Compared to ECDSA, you avoid the computation of a modular inverse. Like ECDSA, you need two scalar-point multiplications (SB and $\mathbf{Hash}(R, A, M)A$).

Ed25519

Ed25519 is a specific instance of EdDSA with the following parameters:

- A *twisted Edwards curve* based on Curve25519, which you'll see in the next section
- SHA-512 as a hash function
- A based point B chosen to optimize efficiency

As of 2023, Ed25519 is likely the second most popular elliptic curve signature algorithm due to its performance benefits and high security guarantees (see *https://ed25519.cr.yp.to*).

OpenSSH, Apple products, and many blockchain platforms use Ed25519 to sign transactions. In February 2023, Ed25519 was added as a NIST standard, as part of FIPS 186-5, "Digital Signature Standard."

Ed25519 has suffered from some interoperability issues, as you'll see later in this chapter.

Encrypting with Elliptic Curves

Although you will typically use elliptic curves for signing, you can encrypt with them. But you'll rarely see people do so in practice because of restrictions on the size of the plaintext that can be encrypted: you can fit about 100 bits of plaintext, as compared to almost 4,000 in RSA with the same security level.

You can encrypt with elliptic curves with the *elliptic curve integrated encryption scheme (ECIES)*, a *hybrid* scheme combining asymmetric and symmetric cryptography. It uses a Diffie–Hellman operation to derive a shared secret key, which protects data with an authenticated cipher.

Given a recipient's public key P, ECIES encrypts a message M, as follows:

1. Pick a random number, d, and compute the point $Q = dG$, where the base point G is a fixed parameter. Here, (d, Q) acts as an ephemeral key pair, used only for encrypting M.
2. Compute an ECDH shared secret by computing $S = dQ$.
3. Use a key derivation function (KDF) to derive a symmetric key, K, from S.
4. Encrypt M using K and a symmetric authenticated cipher, obtaining a ciphertext, C, and an authentication tag, T.

The ECIES ciphertext then consists of the ephemeral public key Q followed by C and T. Decryption is straightforward: the recipient computes S by multiplying Q with their private exponent, then derives the key K, decrypts C, and verifies T.

Choosing a Curve

Criteria to assess the safety of an elliptic curve include the order of the group you use (that is, its number of points), its addition formulas, and the origin of its parameters.

There are several types of elliptic curves, but not all are equal for cryptographic purposes. Choose coefficients a and b in the curve's equation $y^2 = x^3 + ax + b$ carefully, according to established security criteria;

otherwise, you may end up with an insecure curve. In practice, you'll use an established curve for encryption, but knowing what makes a safe curve will help you choose among the several available ones and better understand any associated risks. Here are some points to keep in mind:

- The order of the group shouldn't be a product of small numbers; otherwise, solving ECDLP becomes much easier.
- In "The Addition Law" on page 235, you learned that adding points $P + Q$ required a specific addition formula when $Q = P$. Unfortunately, treating this case differently from the general one may leak critical information if an attacker can distinguish doublings from additions between distinct points. Some curves are secure *because* they use a single formula for all point addition. (When a curve doesn't require a specific formula for doublings, it admits a *unified* addition law.)
- If the creators of a curve don't explain the origin of a and b, they may be suspected of foul play because you can't know whether they've chosen weaker values that enable some yet-unknown attack on the cryptosystem.

Let's review some of the most common curves, especially ones used for signatures or Diffie–Hellman key agreement.

NOTE *You'll find more criteria and details about curves on* https://safecurves.cr.yp.to.

NIST Curves

In 2000, NIST standardized several curves in the FIPS 186 document under "Recommended Elliptic Curves for Federal Government Use." Five NIST curves, called *prime curves*, work modulo a prime number (see "Elliptic Curves over Integers" on page 233). Ten other NIST curves work with binary polynomials, mathematical objects that make implementation in hardware more efficient. (I won't cover binary polynomials in further detail because they're seldom used with elliptic curves.)

The most common NIST curves are the prime curves. Of these, one of the most common is P-256, a curve that works over numbers modulo the 256-bit number $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. The equation for P-256 is $y^2 = x^3 - 3x + b$, where b is a 256-bit number. NIST also provides prime curves of 192 bits, 224 bits, 384 bits, and 521 bits (this isn't a typo; it's 521, not 512).

NIST curves are sometimes criticized because only the NSA, creator of the curves, knows the origin of the b coefficient in their equations. The explanation they've provided is that b results from hashing a random-looking constant with SHA-1. For example, P-256's b parameter comes from the following constant: `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`. No one knows why the NSA picked this particular constant, but most experts don't believe the curve's origin hides any weakness.

Curve25519

Daniel J. Bernstein brought Curve25519 (usually pronounced *curve-twenty-five-five-nineteen*) to the world in 2006. Motivated by performance, he designed Curve25519 to be faster and use shorter keys than the standard curves. But Curve25519 also brings security benefits; unlike the NIST curves, it has no suspicious constants and can use the same unified formula for adding distinct points or for doubling a point.

The form of Curve25519's equation, $y^2 = x^3 + 486662x^2 + x$, is slightly different from the other equations in this chapter, but it still belongs to the elliptic curve family. The unusual form of this equation allows for specific implementation techniques that make Curve25519 fast in software.

Curve25519 works with numbers modulo $2^{255} - 19$, a prime number that's as close as possible to 2^{255}. The b coefficient 486662 is the smallest integer that satisfies the security criteria set by Bernstein. Taken together, these features make Curve25519 more trustworthy than NIST curves and their fishy coefficients.

Curve25519 is used everywhere—for example, in WhatsApp, TLS 1.3, OpenSSH, and many other systems. Following this success, Curve25519 was added to the list of NIST-approved curves in February 2023, as part of the document SP 800-186, "Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters."

NOTE *For the details and rationale behind Curve25519, view the 2016 presentation "The First 10 Years of Curve25519" by Daniel J. Bernstein at* https://cr.yp.to/talks.html#2016.03.09.

Other Curves

As I write this, most cryptographic applications use NIST curves or Curve25519 (including via Ed25519), but there are other legacy standards in use, and newer curves are being promoted and pushed within standardization committees. Old national standards include France's ANSSI curves and Germany's Brainpool curves: two families that don't support complete addition formulas and that use constants of unknown origins.

Some newer curves are more efficient than the older ones and are clear of any suspicion; they offer different security levels and various efficiency optimizations. Examples include Curve41417, a variant of Curve25519, which works with larger numbers and offers a higher level of security (approximately 200 bits); Ed448-Goldilocks, a 448-bit curve first proposed in 2014 and specified in RFC 8032; and six curves proposed by Diego Aranha et al. in "A Note on High-Security General-Purpose Elliptic Curves" (see *https://eprint.iacr.org/2013/647*), though these curves are rarely used. The details specific to these curves are beyond the scope of this book.

Finally, the Ristretto initiative (*https://ristretto.group*) is a technique for constructing prime-order point groups from elliptic curves with non-prime order (and therefore having subgroups). Ristretto offers a safe and

unambiguous way to represent points on an elliptic curve, eliminating risks linked to the structure of Curve25519, for example.

How Things Can Go Wrong

A downside of elliptic curves is their relative complexity. The greater number of parameters than for RSA or classic Diffie–Hellman leaves a larger attack surface and more possibilities for design errors and implementation bugs. ECC implementations can also be vulnerable to side-channel attacks, especially their big-number arithmetic—for example, when the calculation time depends on secret values.

In the following sections, I discuss three examples of vulnerabilities that can occur with elliptic curves, even when the implementation is safe.

ECDSA with Bad Randomness

ECDSA signing is randomized, as it involves a secret random number k when setting $s = (h + rd) / k \bmod n$. However, if you reuse the same k to sign a second message, an attacker could combine the resulting values, $s_1 = (h_1 + rd) / k$ and $s_2 = (h_2 + rd) / k$, to get $s_1 - s_2 = (h_1 - h_2) / k$ and then $k = (h_1 - h_2) / (s_1 - s_2)$. When you know k, you can easily recover the private key d by computing the following:

$$\left(ks_1 - h_1\right) / r = \left(\left(h_1 + rd\right) - h_1\right) / r = rd / r = d$$

Unlike RSA signatures, which won't allow the key to be recovered if using a weak PRNG, the use of nonrandom numbers can lead to ECDSA's k being recoverable, as happened with the attack on the PlayStation 3 game console in 2010, presented by the fail0verflow team at the 27th Chaos Communication Congress in Berlin, Germany. They discovered that the same k was reused to sign different games and were then able to find the signing key, which made it possible to sign any program to authorize it to run on the console.

Invalid Curve Attacks

You can elegantly break ECDH if input points are not properly validated. The primary reason is that the formulas that give the coordinates for the sum of points $P + Q$ never involve the b coefficient of the curve; instead, they rely only on the coordinates of P and Q and the a coefficient (when doubling a point). The unfortunate consequence is that when adding two points, you can't be sure you're working on the right curve because you may actually be adding points on a different curve with a different b coefficient. That means you can break ECDH, as the following scenario, the *invalid curve attack*, describes.

Say that Alice and Bob are running ECDH and agreed on a curve and a base point, G. Bob sends his public key bG to Alice. Alice, instead of sending a public key aG on the agreed upon curve, sends a point on a different curve, either intentionally or accidentally. Unfortunately, this new curve is

weak and allows Alice to choose a point P for which solving ECDLP is easy. She chooses a point of low order, for which there's a relatively small k such that $kP = O$.

Bob, believing that he has a legitimate public key, computes what he thinks is the shared secret bP, hashes it, and uses the resulting key to encrypt data sent to Alice. When Bob computes bP, he's unknowingly computing on the weaker curve. As a result, because P was chosen to belong to a small subgroup within the larger group of points, the result bP also belongs to that small subgroup, allowing an attacker to determine the shared secret bP efficiently if they know the order of P.

One way to prevent this is to confirm points P and Q belong to the right curve by ensuring their coordinates satisfy the curve's equation. This ensures you're able to work on only the secure curve.

Such an invalid curve attack was found in 2015 on certain implementations of the TLS protocol, which uses ECDH to negotiate session keys. (For details, see the paper "Practical Invalid Curve Attacks on TLS-ECDH" by Tibor Jager, Jörg Schwenk, and Juraj Somorovsky.)

Incompatible Ed25519 Validation Rules

You saw that Ed25519 is a signature scheme optimized to offer both efficiency and high security guarantees. The designers of Ed25519 described how it works in detail in a scientific paper and published a well-coded reference implementation. The algorithm was specified as an IETF standard in RFC 8032. One would expect there to be only one version of Ed25519 and different implementations to work in exactly the same way: for a given input value, they should all behave identically and return the same result.

Unfortunately, this is not the case. As cryptographer Henry de Valence documented in the article "It's 255:19AM. Do You Know What Your Validation Criteria Are?" (see *https://hdevalence.ca/blog/2020-10-04-its-25519am*), different implementations of Ed25519 have different criteria for what constitutes a valid signature. Indeed, RFC 8032 doesn't fully describe the validation criteria—and many implementations don't even comply with it.

Of the 15 implementations in de Valence's article, each had its own validation criteria. In particular, the validation of points R (of the signature) and A (the public key) is often different, as is the verification of the equality between $S \times B$ and $R + \mathbf{Hash}(R, A, M) \times A$. Divergences can occur at several levels, including the encoding of points (how you assemble bytes to describe their coordinates) and the validation that points belong to the right subgroup.

As a result, different implementations of the same scheme could behave differently—for example, in a blockchain network where signatures would be accepted by some nodes but rejected by others, a problem for a consensus protocol. All details are explained in de Valence's blog post.

Further Reading

Elliptic curve cryptography is a fascinating, complex topic that involves lots of mathematics. I haven't discussed important notions such as a point's order, a curve's cofactor, projective coordinates, torsion points, and methods for solving the ECDLP problem. If you're mathematically inclined, you'll find information on these and other related topics in the *Handbook of Elliptic and Hyperelliptic Curve Cryptography* by Henri Cohen and Gerhard Frey (Chapman and Hall/CRC, 2005). The 2013 survey "Elliptic Curve Cryptography in Practice" by Joppe Bos, Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow also gives an illustrated introduction with practical examples (*https://eprint.iacr.org/2013/734*).

PART IV

APPLICATIONS

13

TLS

The *Transport Layer Security (TLS) protocol* is the workhorse of internet security. TLS protects connections between servers and clients, whether between a website and its visitors, email servers, a mobile application and its servers, or video game servers and players. Without TLS, the internet wouldn't be very secure.

TLS is application agnostic, meaning you can use it for web-based applications that rely on the HTTP protocol, as well as for any system where a client computer or device needs to initiate a connection with a remote server. For example, TLS is widely used for machine-to-machine communications in internet of things (IoT) applications, such as smart refrigerators that communicate with remote servers.

This chapter provides an abbreviated view of TLS, which has become increasingly complex over the years. Unfortunately, complexity and bloat brought multiple vulnerabilities, and bugs found in its cluttered

implementations have made headlines—Heartbleed, BEAST, CRIME, and POODLE are all vulnerabilities that impacted millions of web servers.

NOTE *You may see folks refer to TLS as* Secure Sockets Layer (SSL)*, which is the name of its predecessor.*

In 2013, engineers started working on TLS 1.3. As you'll learn in this chapter, TLS 1.3 ditched unnecessary and insecure features and replaced old algorithms with then state-of-the-art ones. The result is a simpler, faster, and more secure protocol.

Before we explore how TLS 1.3 works, let's review the problem that TLS aims to solve and the reason for its existence.

Target Applications and Requirements

TLS is the *S* in HTTPS websites, and the padlock that appears in a browser's address bar indicates that a page is secure. The primary driver for creating TLS was to enable secure browsing in applications such as e-commerce or e-banking, by authenticating the site and encrypting traffic to protect sensitive information such as personal data, credit card numbers, and user credentials.

TLS also helps protect general internet-based communication by establishing a *secure channel* between a client and a server that ensures the data transferred is confidential, authenticated, and unmodified.

One of TLS's security goals is to prevent man-in-the-middle attacks, wherein an attacker intercepts encrypted traffic from the transmitting party, decrypts the traffic to capture the clear content, and reencrypts it to send to the receiving party. TLS defeats these attacks by authenticating servers (and optionally clients) using certificates and trusted certificate authorities, as we'll discuss in "Certificates and Certificate Authorities" on page 258.

To ensure wide adoption, TLS needed to satisfy four more requirements: efficiency, interoperability, extensibility, and versatility. For TLS, efficiency means minimizing the performance penalty compared with unencrypted connections. This is good for both the server (to reduce the cost of hardware for the service providers) and the clients (to avoid perceptible delays or the reduction of the battery life of mobile devices). The protocol needed to be interoperable so that it would work on any hardware and any operating system. It was to be extensible so that it could support additional features or algorithms. And it had to be versatile—that is, not bound to a specific application. This parallels the Transport Control Protocol (TCP), which doesn't care about the application protocol used on top of it.

The TLS Protocol Suite

To protect client-server communications, TLS consists of multiple versions of several protocols that form the TLS protocol *suite*. TLS is not a transport protocol and usually sits between the transport protocol (TCP) and an

application layer protocol such as HTTP or SMTP, to secure data transmitted over a TCP connection.

TLS also works over the *User Datagram Protocol (UDP)* transport protocol, which is used for "connectionless" transmissions when latency must be minimal, such as audio or video streaming and online gaming. However, unlike TCP, UDP doesn't guarantee delivery or correct packet ordering. The UDP version of TLS, *Datagram Transport Layer Security (DTLS)*, is therefore slightly different. For more on TCP and UDP, see Charles Kozierok's *The TCP/IP Guide* (No Starch Press, 2005).

The TLS and SSL Families of Protocols

TLS began in 1995 when Netscape developed TLS's ancestor, the SSL protocol. SSL was far from perfect, and both SSL 2.0 and SSL 3.0 had security flaws. You should never use SSL, and you should always use TLS—what adds to the confusion is that people often refer to TLS as SSL, including security experts.

Not all versions of TLS are secure. TLS 1.0 (1999) is the least secure version, though it's still more secure than SSL 3.0. TLS 1.1 (2006) is better but includes a number of weak algorithms. TLS 1.2 (2008) is better yet, but it's complex and provides high security only if configured correctly. Also, its complexity increases the risk of bugs in implementations and the risk of incorrect configurations. For example, TLS 1.2 supports AES in CBC mode, which is often vulnerable to padding oracle attacks.

TLS 1.2 inherited dozens of features and design choices from earlier versions of TLS that make it suboptimal in terms of security and performance. To clean up this mess, cryptography engineers reinvented TLS—keeping only the good parts and adding security features. The result is TLS 1.3, an overhaul that's simplified a bloated design and made it more secure, more efficient, and simpler. Essentially, TLS 1.3 is mature TLS.

TLS in a Nutshell

TLS has two main protocols: the *handshake protocol* (or just *handshake*) determines the secret keys shared between the two parties; the *record protocol* describes how to use these keys to protect data. The data packets TLS processes are called *records*. TLS defines a packet format for encapsulating data from higher-layer protocols for transmission to another party.

The handshake starts with a client that initiates a secure connection with a server. The client sends an initial message called ClientHello with parameters that include the cipher it wants to use. The server checks this message and its parameters and then responds with a ServerHello message. Once the client and the server process each other's messages, they're ready to exchange encrypted data using session keys established through the handshake protocol, as you'll see in "The TLS Handshake Protocol" on page 263.

Certificates and Certificate Authorities

The most critical step in the TLS handshake, and the crux of TLS's security, is the *certificate validation step*, wherein a server uses a certificate to authenticate itself to a client.

A *certificate* is essentially a public key accompanied by a signature of that key and associated information (including the domain name). For example, when connecting to *https://www.google.com*, your browser receives a certificate from some network host and then verifies the certificate's signature, which reads something like "I am *google.com*, and my public key is [*key*]." If the signature is verified, the certificate and its public key are *trusted*, and the browser proceeds with establishing the connection. (See Chapters 10 and 12 for details regarding signatures.)

The browser knows the public key needed to verify the signature through a *certificate authority (CA)*, which is essentially a public key hardcoded in your browser or operating system. The public key's private key (that is, its signing capability) belongs to a trusted organization that ensures the public keys in certificates it issues belong to the website or entity that claims them. That is, a CA acts as a *trusted third party*. Without CAs, there's no way to verify that the public key served by *google.com* belongs to Google and not to an eavesdropper performing a man-in-the-middle attack.

For example, Listing 13-1 shows what happens when you use the OpenSSL command line tool to initiate a TLS connection to *www.google.com* on port 443, the network port used for TLS-based HTTP connections (HTTPS).

```
$ openssl s_client -connect www.google.com:443
CONNECTED(00000003)
---
Certificate chain
 0 s:CN = www.google.com
   i:C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
 1 s:C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
   i:C = US, O = Google Trust Services LLC, CN = GTS Root R1
 2 s:C = US, O = Google Trust Services LLC, CN = GTS Root R1
   i:C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIEiDCCA3CgAwIBAgIQNJvIv7ypW9IQHaA7P69MzzANBgkqhkiG9w0BAQsFADBG
MQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIExM
QzETMBEGA1UEAxMKR1RTIENBIDFDMzAeFw0yMzA5MDQwODIzMjlaFw0yMzExMjcw
ODIzMjhaMBkxFzAVBgNVBAMTDnd3dy5nb29nbGUuY29tMFkwEwYHKoZIzj0CAQYI
KoZIzj0DAQcDQgAENEMvWAYOTRTbOw5ZxbUbX/Z+EcviE5OSzQzvP/xyyVIaURM4
AOJer9IJO/6Iq6o2AfDXUxrdBKpSzlzaeFCaqqOCAmgwggJkMA4GA1UdDwEB/wQE
AwIHgDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMBAf8EAjAAMBOGA1UdDgQW
BBRnwQnVC8ok1e6YPIbQwjB+XFpPRjAfBgNVHSMEGDAWgBSKdH+vhc3ulc09nNDi
RhTzcTUdJzBqBggrBgEFBQcBAQReMFwwJwYIKwYBBQUHMAGGG2hOdHA6Ly9vY3Nw
LnBraS5nb29nL2dOczFjMzAxBggrBgEFBQcwAoYlaHR0cDovL3BraS5nb29nL3Jl
cG8vY2VydHMvZ3RzMWMzLmRlcjAZBgNVHREEEjAQgg53d3cuZ29vZ2xlLmNvbTAh
BgNVHSAEGjAYMAgGBmeBDAECATAMBgorBgEEAdZ5AgUDMDwGA1UdHwQ1MDMwMaAv
```

```
oC2GK2hOdHA6Ly9jcmxzLnBraS5nb29nL2dOczFjMy9mVkp4YlYtS3Rtay5jcmww
ggEFBgorBgEEAdZ5AgQCBIH2BIHzAPEAdgDoPtDaPvUGNTLnVyi8iWvJA9PLORFr
7Otp4Xd9bQa9bgAAAYpfgRjOAAAEAwBHMEUCID+BcS984SEh2E2UrKfLvF2fG7qa
SYkzbELytrDz91wmAiEAlwOPMM26CynmadsqomPMXKdRNMvzdyciHVimhOsnrBAA
dwB6MoxU2LcttiDqOOBSHumEFnAyE4VNO9IrwTpXo1LrUgAAAYpfgRkAAAAEAwBI
MEYCIQDZf1tULkVCXRc68zJwgp5WFJUbTxFjz6CP+eLb3dgz3gIhAJ9uS7psu2Gl
HdTXokXTetMY7MCdIcuj6OQm/qTn+1dFMAOGCSqGSIb3DQEBCwUAA4IBAQCIEnOv
QzaqNCOhiI5TKcRhaR24yKid3F57a/GOM1LDE/v7oCm+3fxtvuK9HVa/Dmnvavqp
ci7TpMDj/ocXjE4dL4/yHaVx6GhTDKMW/bbBkDaqXoSdb9lAcUZPLRTV4AjFdjmB
8wZTf95bnfeuKNXWlbo/k/9pRRhFNKKMUI54xLiVdj4wk1EUAsrMTTn+Ol2ZbFeS
s614abBT5WOhhFkLvjvEht8p3UKQwwyhRjZMBsae/dOQfT8hglVtVhhGd7f1hFqI
XSERl8EsyDc3urCsa+RjUjCvE9Q+y2X8WVVOHPCjwsANU56qEZmh4kqgg5paW/SL
maM/8Vny/nKNd6Dj
-----END CERTIFICATE-----
subject=CN = www.google.com

issuer=C = US, O = Google Trust Services LLC, CN = GTS CA 1C3

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 4295 bytes and written 396 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Listing 13-1: Establishing a TLS connection with www.google.com *and receiving certificates to authenticate the connection*

The certificate data is between the `BEGIN CERTIFICATE` and `END CERTIFICATE` markers. Before this, the `Certificate chain` contains a description of the certificate chain, wherein the lines beginning with `s:` describe the *subject* of the certified entity, and the lines beginning with `i:` describe the *issuer* of the signature. Certificate number 0 is received by *www.google.com*, certificate 1 belongs to the entity that signed certificate 0, and certificate 2 belongs to the entity that signed certificate 1.

The organization that issued certificate 0 is Google (via the Google Trust Services entity, GTS), which authorized the issuance of certificate 0 for the domain name *www.google.com* by signing the certificate with the private key GTS CA 1C3. The certificate attesting that this key belongs to the Google key hierarchy is certificate 1, which is signed by the GTS key Root R1, a root certificate within Google. Certificate 2, issued by GlobalSign

(a recognized certification authority), attests that the GTS Root R1 key belongs to the Google organization.

In this example, your operating system usually already has certificates 1 and 2, which it considers trusted certificates. In such a case, you just need to check two signatures: Google's GTS CA 1C3 entity in certificate 0 and Google's GTS Root R1 entity in certificate 1. If your system doesn't already include certificate 2 as a trusted certificate but has GlobalSign's root certificate (GlobalSign Root CA), then you'll also need to check GlobalSign's signature in certificate 2.

Certificate authority organizations such as Google and GlobalSign must be trustworthy and issue certificates only to trustworthy entities, and they must protect their private keys in order to prevent an attacker from issuing certificates on their behalf (for example, to impersonate a legitimate *www.google.com* server).

To see what's in a certificate, enter the command **`openssl x509 -text -noout`** in a Unix terminal and then paste the certificate in Listing 13-1. The output appears in Listing 13-2.

```
$ openssl x509 -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      34:9b:c8:bf:bc:a9:5b:d2:10:1d:a0:3b:3f:af:4c:cf
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
    Validity
      Not Before: Sep  4 08:23:29 ...
      Not After : Nov 27 08:23:28 ...
    Subject: CN = www.google.com
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
        Public-Key: (256 bit)
        pub:
          04:34:43:2f:58:06:34:4d:14:db:d3:0e:59:c5:b5:
          1b:5f:f6:7e:11:cb:e2:13:9d:12:cd:0c:ef:3f:fc:
          72:c9:52:1a:51:13:38:03:42:5e:af:d2:09:3b:fe:
          88:ab:aa:36:01:f0:d7:53:1a:dd:04:aa:52:ce:5c:
          da:78:50:9a:aa
        ASN1 OID: prime256v1
        NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Extended Key Usage:
        TLS Web Server Authentication
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Subject Key Identifier:
        67:C1:09:D5:0B:CA:24:D5:EE:98:3C:86:D0:C2:30:7E:5C:5A:4F:46
      X509v3 Authority Key Identifier:
        8A:74:7F:AF:85:CD:EE:95:CD:3D:9C:D0:E2:46:14:F3:71:35:1D:27
```

```
        Authority Information Access:
          OCSP - URI:http://ocsp.pki.goog/gts1c3
          CA Issuers - URI:http://pki.goog/repo/certs/gts1c3.der
        X509v3 Subject Alternative Name:
          DNS:www.google.com
        X509v3 Certificate Policies:
          Policy: 2.23.140.1.2.1
          Policy: 1.3.6.1.4.1.11129.2.5.3
        X509v3 CRL Distribution Points:
          Full Name:
            URI:http://crls.pki.goog/gts1c3/fVJxbV-Ktmk.crl
        CT Precertificate SCTs:
          Signed Certificate Timestamp:
            Version   : v1 (0x0)
            Log ID    : E8:3E:D0:DA:3E:F5:06:35:32:E7:57:28:BC:89:6B:C9:
                        03:D3:CB:D1:11:6B:EC:EB:69:E1:77:7D:6D:06:BD:6E
            Timestamp : Sep  4 09:23:30.638 2023 GMT
            Extensions: none
            Signature : ecdsa-with-SHA256
              30:45:02:20:3F:81:71:2F:7C:E1:21:21:D8:4D:94:AC:
              A7:CB:BC:5D:9F:1B:BA:9A:49:89:33:6C:42:F2:B6:B0:
              F3:F7:5C:26:02:21:00:97:03:8F:30:CD:BA:0B:29:E6:
              69:DB:2A:A2:63:CC:5C:A7:51:34:CB:F3:77:27:22:1D:
              58:A6:87:4B:27:AC:10
          Signed Certificate Timestamp:
            Version   : v1 (0x0)
            Log ID    : 7A:32:8C:54:D8:B7:2D:B6:20:EA:38:E0:52:1E:E9:84:
                        16:70:32:13:85:4D:3B:D2:2B:C1:3A:57:A3:52:EB:52
            Timestamp : Sep  4 09:23:30.688 2023 GMT
            Extensions: none
            Signature : ecdsa-with-SHA256
              30:46:02:21:00:D9:7F:5B:54:2E:45:42:5D:17:3A:F3:
              32:70:82:9E:56:14:95:1B:4F:11:63:CF:A0:8F:F9:E2:
              DB:DD:D8:33:DE:02:21:00:9F:6E:4B:BA:6C:BB:61:A5:
              1D:D4:D7:A2:45:D3:7A:D3:18:EC:C0:9D:21:CB:A3:EB:
              44:26:FE:A4:E7:FB:57:45
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
      88:12:7d:2f:43:36:aa:34:23:a1:88:8e:53:29:c4:61:69:1d:
      b8:c8:a8:9d:dc:5e:7b:6b:f1:8e:33:52:c3:13:fb:fb:a0:29:
      be:dd:fc:6d:be:e2:bd:1d:56:bf:0e:69:ef:6a:fa:a9:72:2e:
      d3:a4:c0:e3:fe:87:17:8c:4e:1d:2f:8f:f2:1d:a5:71:e8:68:
      53:0c:a3:16:fd:b6:c1:90:36:aa:5e:84:9d:6f:d9:40:71:46:
      4f:2d:14:d5:e0:08:c5:76:39:81:f3:06:53:7f:de:5b:9d:f7:
      ae:28:d5:d6:95:ba:3f:93:ff:69:45:18:45:34:a2:8c:50:8e:
      78:c4:b8:95:76:3e:30:93:51:14:02:ca:cc:4d:39:fe:3a:5d:
      99:6c:57:92:b3:ad:78:69:b0:53:e5:6d:21:84:59:0b:be:3b:
      c4:86:df:29:dd:42:90:c3:0c:a1:46:36:4c:06:c6:9e:fd:dd:
      10:7d:3f:21:82:55:6d:56:18:46:77:b7:f5:84:5a:88:5d:21:
      11:97:c1:2c:c8:37:37:ba:b0:ac:6b:e4:63:52:30:af:13:d4:
      3e:cb:65:fc:59:55:74:1c:f0:a3:c2:c0:0d:53:9e:aa:11:99:
      a1:e2:4a:a0:83:9a:5a:5b:f4:8b:99:a3:3f:f1:59:f2:fe:72:
      8d:77:a0:e3
```

Listing 13-2: Decoding a certificate received from www.google.com

This listing shows the command `openssl x509` decoding a certificate, originally provided as a block of base64-encoded data. Because OpenSSL knows the structure of this data, it can tell you what's inside the certificate, including a serial number and version information, identifying information, validity dates (the `Not Before` and `Not After` lines), a public key (here as an RSA modulus and its public exponent), and a signature of the preceding information.

Although security experts and cryptographers often claim the whole certificate system is inherently broken, it's one of the best solutions we have, along with the trust-on-first-use (TOFU) policy adopted by SSH, for example.

The Record Protocol

All data exchanged through TLS 1.3 communications is transmitted as sequences of TLS records, the data packets used by TLS. The TLS record protocol (the *record layer*) is essentially a transport protocol, agnostic of the transported data's meaning; this makes TLS suitable for any application.

The TLS record protocol first carries the data exchanged during the handshake. Once the handshake is complete and both parties share a secret key, application data is fragmented into chunks that transmit as part of the TLS records.

The Structure of a TLS Record

A TLS record is a chunk of data of at most 16KB with the following structure:

- The first byte represents the type of data transmitted and is set to the value `22` for handshake data, `23` for encrypted data, and `21` for alerts. The TLS 1.3 specifications call this value ContentType.
- The second and third bytes are set to `03` and `01`, respectively. These bytes are fixed for historical reasons and aren't unique to TLS version 1.3. The specifications call this 2-byte value ProtocolVersion.
- The fourth and fifth bytes encode the length of the data to transmit as a 16-bit integer, which can be no larger than 2^{14} bytes (16KB).
- The rest of the bytes are the data to transmit (or the *payload*), of a length equal to the value encoded by the record's fourth and fifth bytes.

NOTE *A TLS record has a relatively simple structure. As you've seen, a TLS record's header includes only three fields. For comparison, an IPv4 packet includes 14 fields before its payload, and a TCP segment includes 13 fields.*

When the first byte of a TLS 1.3 record (ContentType) is set to `23`, an authenticated cipher encrypts and authenticates its payload. The payload consists of a ciphertext followed by an authentication tag, which the receiving end, respectively, decrypts and verifies. The recipient knows which

cipher and key to decrypt with, thanks to the magic of TLS: if you receive an encrypted TLS record, you already know the cipher and key because executing the handshake protocol establishes them.

Nonces

Unlike many other protocols, such as IPsec's Encapsulating Security Payload (ESP), TLS records don't specify the nonce the authenticated cipher will use.

The nonces that encrypt and decrypt TLS records are derived from 64-bit sequence numbers, maintained locally by each party and incremented for each new record. When the client encrypts data, it derives a nonce by XORing the sequence number with a `client_write_iv` value, itself derived from the shared secret. The server uses a similar method to choose nonces when transmitting data, but with a `server_write_iv` value.

For example, if you transmit three TLS records, you'll derive a nonce from 0 for the first record, from 1 for the second, and from 2 for the third; if you then receive three records, you'll also use nonces 0, 1, and 2, in this order. Reusing the same sequence numbers values for encrypting transmitted data and decrypting receiving data isn't a weakness because they're XORed with different constants (`client_write_iv` and `server_write_iv`) and because you use different secret keys for each direction.

The Zero Padding Feature

TLS 1.3 records support *zero padding*, which mitigates traffic analysis attacks. Attackers use *traffic analysis* to extract information from traffic patterns using timing, volume of data transferred, and so on. For example, because ciphertexts are approximately the same size as plaintexts, even when using strong encryption, attackers can determine the approximate size of messages by simply looking at the length of their ciphertext.

Zero padding adds zeros to the plaintext to inflate the ciphertext's size, fooling observers into thinking that an encrypted message is longer than it really is.

The TLS Handshake Protocol

The handshake is the crux of the TLS agreement protocol—the process by which a client and a server establish shared secret keys to initiate secure communications. During a TLS handshake, the client and the server play different roles. The client proposes some configurations (the TLS version and a suite of ciphers, in order of preference), and the server chooses the configuration it will use. The server should follow the client's preferences. To ensure interoperability between implementations and that any server implementing TLS 1.3 will be able to read TLS 1.3 data sent by any client implementing TLS 1.3 (even if it's using a different library or programming language), the TLS 1.3 specifications also describe the format data should be sent in.

Figure 13-1 shows how the handshake process exchanges data, as the TLS 1.3 specifications describe.

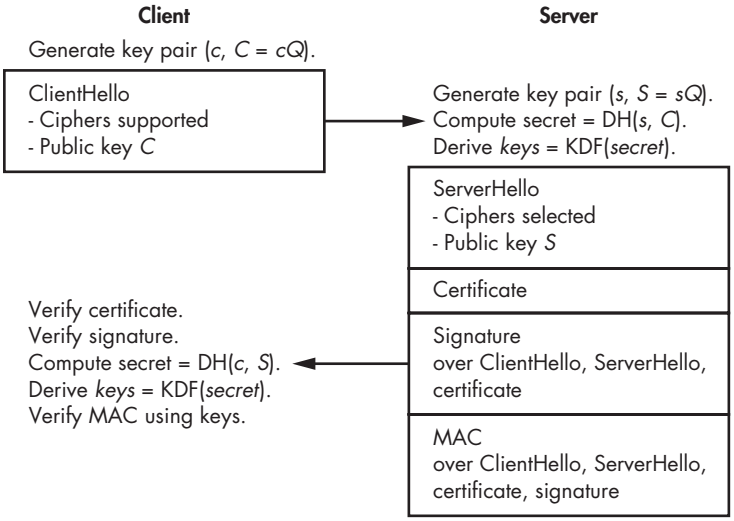


Figure 13-1: The TLS 1.3 handshake process

In the TLS 1.3 handshake, the client sends a message to the server saying, "I want to establish a TLS connection with you. Here are the ciphers that I support to encrypt TLS records, and here is a Diffie–Hellman public key." The public key must be generated specifically for this TLS session, and the client keeps the associated private key. The message sent by the client also includes a 32-byte random value and optional information (such as additional parameters). This first message, *ClientHello*, must follow a specific format when transmitted as a series of bytes, as the TLS 1.3 specification defines.

The server receives the ClientHello message, verifies that it's correctly formatted, and responds with a ServerHello message, which is loaded with information. Typically, when connecting to an HTTPS website, it contains the cipher that will encrypt TLS records, a Diffie–Hellman public key, a 32-byte random value (discussed in "Downgrade Protection" on page 266), a certificate, a signature of all the previous information in ClientHello and ServerHello messages (computed using the private key associated with the certificate's public key), and a MAC of that same information, plus the signature. The MAC is computed using a symmetric key derived from the Diffie–Hellman shared secret, which the server computes from its Diffie–Hellman private key and the client's public key.

When the client receives the ServerHello message, it verifies the certificate's validity, verifies the signature, computes the shared Diffie–Hellman secret and derives symmetric keys from it, and verifies the MAC sent by the server. Once everything is verified, the client is ready to send encrypted messages to the server.

NOTE *TLS 1.3 supports many options and extensions, so it may behave differently. You can, for example, configure the TLS 1.3 handshake to require a client certificate so that the server verifies the identity of the client. TLS 1.3 also supports a handshake with preshared keys.*

Let's look at this in practice. Say you've deployed TLS 1.3 to provide secure access to the website *https://www.nostarch.com*. When you point your browser (the client) to this site, your browser sends a ClientHello message to the site's server that includes the ciphers that it supports. The website responds with a ServerHello message and a certificate that includes a public key associated with the domain *www.nostarch.com*. The client verifies the certificate's validity using one of the certificate authorities embedded in the browser (a trusted certificate authority, whose certificate should be included in the browser's or the operating system's certificate store to be validated, should sign the received certificate). Once all checks pass, the browser requests the site's initial page from the *www.nostarch.com* server.

Upon a successful TLS 1.3 handshake, all communications between the client and the server are encrypted and authenticated. An eavesdropper can learn that a client at a given IP address is talking to a server at another given IP address and can observe the encrypted content exchanged but can't learn the underlying plaintext or modify the encrypted messages (if they do, the receiving party will notice that the communication has been tampered with, because messages are authenticated). That's enough security for many applications.

TLS 1.3 Cryptographic Algorithms

TLS 1.3 uses authenticated encryption algorithms, a key derivation function (a hash function that derives secret keys from a shared secret), as well as a Diffie–Hellman operation—but how exactly do these work, what algorithms are used, and how secure are they?

With regard to the choice of authenticated ciphers, TLS 1.3 supports only three algorithms: AES-GCM, AES-CCM (a slightly less efficient mode than GCM), and the ChaCha20 stream cipher combined with the Poly1305 MAC (as defined in RFC 7539). Because TLS 1.3 prevents using an unsafe key length such as 64 or 80 bits, the secret key can be either 128 bits (AES-GCM or AES-CCM) or 256 bits (AES-GCM or ChaCha20-Poly1305).

The key derivation operation (KDF) in Figure 13-1 is based on HKDF, a construction based on HMAC (see Chapter 7) and defined in RFC 5869 that uses either the SHA-256 or the SHA-384 hash function.

Your options for performing the Diffie–Hellman operation (the core of the TLS 1.3 handshake) are limited to elliptic curve cryptography and a multiplicative group of integers modulo a prime number (as in traditional Diffie–Hellman). But you can't use just any elliptic curve or group: the supported curves include three NIST curves as well as Curve25519 (see Chapter 12) and Curve448, both defined in RFC 7748. TLS 1.3 also supports DH over groups of integers, as opposed to elliptic curves. The groups

supported are the five groups defined in RFC 7919: groups of 2,048, 3,072, 4,096, 6,144, and 8,192 bits.

The 2,048-bit group may in theory be TLS 1.3's weakest link. Whereas the other options provide at least 128-bit security, 2,048-bit Diffie–Hellman is believed to provide less than 100-bit security. Supporting a 2,048-bit group can therefore be seen as inconsistent with other TLS 1.3 design choices. In practice, 100-bit security is about as hard to crack as 128-bit—that is, practically impossible.

TLS 1.3 Improvements over TLS 1.2

TLS 1.3 is very different from its predecessor. For one, it gets rid of weak algorithms like MD5, SHA-1, RC4, and AES in CBC mode. Also, whereas TLS 1.2 often protected records using a combination of a cipher and a MAC (such as HMAC-SHA-1) within a MAC-then-encrypt construction, TLS 1.3 supports only the more efficient and secure authenticated ciphers. TLS 1.3 also ditches elliptic curve point encoding negotiation and defines a single point format for each curve.

TLS 1.3 removed features in 1.2 that weakened the protocol, and it reduced the protocol's overall complexity and thereby its attack surface. For example, TLS 1.3 ditches optional data compression, a feature that enabled the CRIME attack on TLS 1.2. This attack exploited the fact that the length of the compressed version of a message leaks information on the content of the message.

But TLS 1.3 also brings new features that make connections either more secure or more efficient. I'll discuss three of these features: downgrade protection, the single round-trip handshake, and session resumption.

Downgrade Protection

TLS 1.3's *downgrade protection* feature is a defense against *downgrade attacks*, wherein an attacker forces the client and the server to use a weaker version of TLS than 1.3. To carry out a downgrade attack, an attacker forces the server to use a weaker version of TLS by intercepting and modifying the ClientHello message to tell the server that the client doesn't support TLS 1.3. Now the attacker can exploit vulnerabilities in earlier versions of TLS.

In an effort to defeat downgrade attacks, the TLS 1.3 server uses three types of patterns in the 32-byte random value sent within the ServerHello message to identify the type of connection requested. The pattern should match the client's request for a specific type of TLS connection. If the client receives the wrong pattern, it knows something is up.

Specifically, if the client asks for a TLS 1.2 connection, the first 8 of the 32 bytes are set to `44 4F 57 4E 47 52 44 01`, and if it asks for a TLS 1.1 connection, they're set to `44 4F 57 4E 47 52 44 00`. However, if the client requests a TLS 1.3 connection, these first 8 bits should be random. For example, if a client sends a ClientHello asking for a TLS 1.3 connection, but an

attacker on the network modifies it to ask for a TLS 1.1 connection, when the client receives the ServerHello with the wrong pattern, it knows that its ClientHello message was modified. (The attacker can't arbitrarily modify the server's 32-byte random value because this value is cryptographically signed.)

Single Round-Trip Handshake

In a typical TLS 1.2 handshake, the client sends some data to the server, waits for a response, and then sends more data and waits for the server's response before sending encrypted messages. The delay is that of two round-trip times (RTT). In contrast, TLS 1.3's handshake takes a single round-trip time (see Figure 13-1). The time saved can be in the hundreds of milliseconds. This is significant when you consider that servers of popular services handle thousands of connections per second.

Session Resumption

TLS 1.3 is faster than TLS 1.2, but it can be made even faster (on the order of hundreds of milliseconds) by completely eliminating the round trips that precede an encrypted session. The trick is to use *session resumption*, which leverages the preshared key exchanged between the client and the server in a previous session to bootstrap a new session. Session resumption brings two major benefits: the client can start encrypting immediately, and there's no need to use certificates in subsequent sessions.

Figure 13-2 shows how session resumption works.

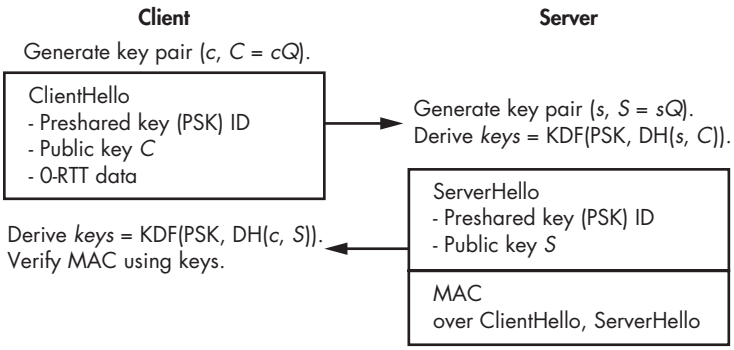


Figure 13-2: The TLS 1.3 session resumption handshake, wherein the 0-RTT data is the session resumption data sent along with the ClientHello

First, the client sends a ClientHello message that includes the identifier of the key already shared (called *PSK* for *preshared key*) with the server, along with a fresh DH public key. The client can also include encrypted data in this first message (called *0-RTT data*). When the server responds to a ClientHello message, it provides a MAC over the data exchange. The client verifies the MAC and knows that it's talking to the same server as it did previously, thus rendering certificate validation somewhat superfluous. The

client and the server perform a Diffie–Hellman key agreement as in the normal handshake, and subsequent messages are encrypted using keys that depend on both the PSK and the newly computed Diffie–Hellman shared secret.

The Strengths of TLS Security

We'll evaluate the strengths of TLS 1.3 with respect to two main security notions from Chapter 11: authentication and forward secrecy.

Authentication

During the TLS 1.3 handshake, the server authenticates to the client using the certificate mechanism. However, the client isn't authenticated, and clients may authenticate with a server-based application (such as Gmail) by providing a username and password in a TLS record after performing the handshake. If the client's already established a session with the remote service, it may authenticate by sending a *secure cookie*, which can be sent only through a TLS connection.

In certain cases, clients can authenticate to a server using a certificate-based mechanism similar to what the server uses to authenticate to the client: the client sends a *client certificate* to the server, which verifies this certificate before authorizing the client. However, you will rarely use client certificates because they complicate things for both clients and the server (that is, the certificate issuer): clients need to perform complex operations to integrate the certificate into their system and to protect its private key, while the issuer needs to make sure that only authorized clients received a certificate, among other requirements.

Forward Secrecy

Recall from "Key Agreement Protocols" in Chapter 11 that a key agreement provides forward secrecy if previous sessions aren't compromised when the present session is compromised. In the data leak model, only temporary secrets are compromised, whereas in the breach model, long-term secrets are exposed.

Thankfully, TLS 1.3 forward secrecy holds up in the face of both a data leak and a breach. In the data leak model, the attacker recovers temporary secrets such as the session keys or Diffie–Hellman private keys of a specific session (the values *c*, *s*, *secret*, and *keys* in Figure 13-1). However, they can use these values to decrypt communications from only the present session, not previous sessions, because different values of *c* and *s* were used (thus yielding different keys).

In the breach model, the attacker also recovers long-term secrets (namely, the private key that corresponds to the public key in the certificate). However, this is no more useful when decrypting previous sessions than temporary secrets, because this private key serves to authenticate only the server, and forward secrecy holds up again.

In practice, if an attacker compromises a client's machine and gains access to all of its memory, they may recover the client's TLS session keys and secrets for the current session from memory. But more importantly, if previous keys are still in memory, the attacker may be able to find them and decrypt previous sessions, thereby bypassing the theoretical forward secrecy. Therefore, for a TLS implementation to ensure forward secrecy, it must properly erase keys from memory once they're no longer used, typically by zeroing out the memory.

How Things Can Go Wrong

TLS 1.3 fits the bill as a general-purpose secure communications protocol, but it's not bulletproof. Like any security system, it can fail under certain circumstances (for example, when the assumptions made by its designers about real attacks are wrong). Unfortunately, even the latest version of TLS 1.3, configured with the most secure ciphers, can be compromised. For example, TLS 1.3 security relies on the assumption that all three parties (the client, the server, and the certificate authority) will behave honestly, but what if one party is compromised or the TLS implementation itself is poorly implemented?

Compromised Certificate Authority

Root certificate authorities (root CAs) are organizations that browsers trust to validate certificates served by remote hosts. For example, if your browser accepts the certificate provided by *www.google.com*, the assumption is that a trusted CA has verified the legitimacy of the certificate owner. The browser verifies the certificate by checking its CA-issued signature. Since only the CA knows the private key required to create this signature, we assume others can't create valid certificates on behalf of the CA. Very often a website's certificate won't be signed by a root CA but by an intermediate CA, which is connected to the root CA through a certificate chain.

If a CA's private key is compromised, the attacker is able to use the CA's private key to create a certificate for any URLs in, say, the *google.com* domain without Google's approval. The attacker can then use those certificates to pretend to host a legitimate server or subdomain like *mail.google.com* and intercept a user's credentials and communications. That's exactly what happened in 2011 when an attacker hacked into the network of the Dutch certificate authority DigiNotar and created seemingly legitimate certificates. The attacker used these fake certificates for several Google services.

Compromised Server

If a server is compromised and fully controlled by an attacker, all is lost: the server holds the session keys, being the termination point of the TLS connection. The attacker can see all transmitted data before it's encrypted and all received data once it's decrypted. They'll also likely get their hands on the server's private key, which could allow them to impersonate the

legitimate server using their own malicious server. TLS won't save you in this case.

Fortunately, such security disasters are rarely seen in high-profile applications such as Gmail and iCloud, which are well protected and sometimes have their private keys stored in a separate security module, such as a hardware security module (HSM), directly or via a key management system (KMS) application.

Attacks on web applications via vulnerabilities such as database query injections and cross-site scripting are more common because they're mostly independent of TLS's security and are carried out by attackers over a legitimate TLS connection. Such attacks may compromise usernames, passwords, and so on.

Compromised Client

TLS security is also jeopardized when a client, such as a browser, is compromised by a remote attacker. Having compromised the client, the attacker is able to capture session keys, read any decrypted data, and so on. They could even install a rogue CA certificate in the client's system to have it silently accept otherwise-invalid certificates, thereby letting attackers intercept TLS connections.

The difference between the compromised CA or server scenarios and the compromised client scenario is that in the case of the compromised client, only the targeted client is affected, instead of potentially *all* clients.

Bugs in Implementations

As with any cryptographic component, TLS can fail when there are bugs in its implementation. The poster child for TLS bugs is Heartbleed (see Figure 13-3), a buffer overflow in the OpenSSL implementation of a minor TLS feature called *heartbeat*. Heartbleed was discovered in 2014, independently by a Google researcher and by the Codenomicon company, and affected millions of TLS servers and clients.

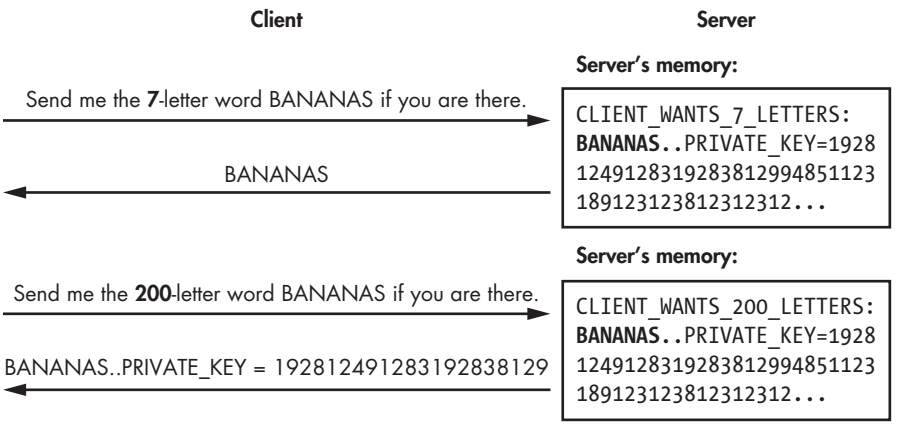


Figure 13-3: The Heartbleed bug in OpenSSL implementations of TLS

A client first sends a buffer along with a buffer length to the server to check whether the server is online. In this example, the buffer is the string *BANANAS*, and the client explicitly says that this word is seven letters long. The server reads the seven-letter word and returns it to the client.

The problem is that the server doesn't confirm that the length is correct and attempts to read as many characters as the client tells it to. Consequently, if the client provides a length that is longer than the string's actual length, the server reads too much data from memory and returns it to the client, together with any extra data that may contain sensitive information, such as private keys or session cookies.

The Heartbleed bug came as a shock. To avoid similar future bugs, OpenSSL and other major TLS implementations now perform rigorous code reviews and use automated tools such as fuzzers to identify potential issues.

Further Reading

This chapter isn't a comprehensive guide to TLS, and you may want to dig deeper into the history of TLS, its previous vulnerabilities, and its latest version. The complete TLS 1.3 specifications, found on the home page of the TLS Working Group (TLSWG) at *https://tlswg.org*, include everything about the protocol (though not necessarily its underlying rationale).

I also suggest you learn about major protocols that use TLS, such as QUIC (used in connections between Chrome and Google servers) and SRTP (used for videoconferencing and streaming traffic).

In addition, here are two important initiatives related to TLS deployment:

- SSL Labs TLS test (*https://www.ssllabs.com/ssltest*) is a free service by Qualys that lets you test a browser's or a server's TLS configuration, providing a security rating as well as improvement suggestions. If you set up your own TLS server, use this test to make sure everything is safe.
- Let's Encrypt (*https://letsencrypt.org*) is a nonprofit that offers a service to "automagically" deploy TLS on your HTTP servers. It includes features to automatically generate a certificate and configure the TLS server, and it supports all the common web servers and operating systems.

14

QUANTUM AND POST-QUANTUM

In this chapter, we'll examine the future of cryptography over a time horizon of, say, a century or more—one in which quantum computers may exist. Quantum computers leverage phenomena from quantum physics to run different kinds of algorithms than we're used to. While large quantum computers don't exist yet, they have the potential to break RSA, Diffie–Hellman, and elliptic curve cryptography—all the public-key crypto deployed or standardized as of this writing.

To ensure against the quantum computing risk, cryptography researchers have developed alternative public-key *post-quantum* algorithms. In 2015, the NSA called for a transition to quantum-resistant algorithms, and in 2017 NIST began a process to standardize post-quantum algorithms, which they announced as part of new standards in 2022.

This chapter provides a nontechnical overview of the principles behind quantum computers as well as a glimpse of post-quantum algorithms. There's some math involved, but only basic arithmetic and linear algebra, so don't fret about the unusual notations.

How Quantum Computers Work

Quantum computing uses quantum physics to compute differently and perform tasks that classical computers can't, such as breaking RSA and elliptic curve cryptography efficiently. A quantum computer is not a superfast normal computer. In fact, quantum computers can't efficiently solve any problem that's too hard for a classical computer, such as brute-force search or **NP**-hard problems.

Quantum mechanics—the branch of physics that studies the behavior of subatomic particles, which behave truly randomly—is the basis of quantum computers. Unlike classical computers, which operate on bits that are either 0 or 1, quantum computers operate on *quantum bits* (or *qubits*), which can be "both 0 and 1 simultaneously." This is a state of ambiguity called *superposition*, where my quotation marks indicate an oversimplification. In this microscopic world, physicists discovered that particles such as electrons and photons behave in a highly counterintuitive way: before you observe an electron, it isn't at a definite location in space but in several locations at the same time (that is, in a state of superposition). Once you observe it—an operation called *measurement*—it stops at a fixed, random location and is no longer in superposition; the quantum state *collapsed*. This enables the creation of qubits in a quantum computer, along with the phenomena of entanglement and interference.

Quantum computers work because of *entanglement*, wherein two particles are connected (entangled) in a way that observing the value of one gives the value of the other, even if the two particles are widely separated (kilometers or even light-years away from each other). The *Einstein–Podolsky–Rosen (EPR) paradox* illustrates this behavior, which caused Albert Einstein to initially dismiss quantum mechanics. (See *https://plato.stanford.edu/entries/qt-epr/* for an in-depth explanation.)

Interference is also crucial to the operation of quantum computers. With this property, particles can combine or cancel out each other's effects due to their wave-like nature, as the double-slit experiment famously illustrates. Quantum computing exploits interference so that the "waves" of valid solutions reinforce each other and invalid solutions cancel each other out.

To explain how a quantum computer works, I'll distinguish the actual quantum computer (the hardware, including its quantum bits) from quantum algorithms (the software that runs on it, composed of *quantum gates*).

Quantum Bits

You can characterize qubits, or groups thereof, with *amplitudes*, which are numbers akin to probabilities that aren't *exactly* probabilities. Whereas a

probability is a number between 0 and 1, an amplitude is a complex number of the form $a + bi$ (that is, $a + b \times i$), where a and b are real numbers and i is an *imaginary unit*. You use the number i to form *imaginary numbers* of the form bi, with b a real number. When you multiply i by a real number, you get another imaginary number, and multiplying it by itself it results in –1 (that is, $i^2 = -1$).

Unlike real numbers, which you see as belonging to a line (see Figure 14-1), *complex numbers* belong to a plane (a space with two dimensions), as Figure 14-2 demonstrates.

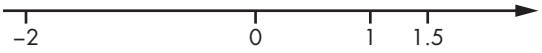


Figure 14-1: A view of real numbers as points on an infinite straight line

In Figure 14-2, the x-axis corresponds to the a in $a + bi$, the y-axis corresponds to the b, and the dotted lines correspond to the real and imaginary parts of each number. For example, the vertical dotted line going from the point $3 + 2i$ down to 3 is two units long (the 2 in the imaginary part $2i$).

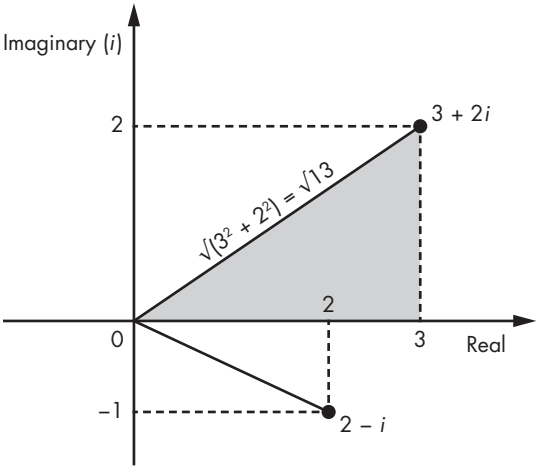


Figure 14-2: A view of complex numbers as points in a two-dimensional space

You can use the Pythagorean theorem to compute the length of the line segment going from the origin (0) to the point $a + bi$ by viewing this line as the hypotenuse of a triangle. The hypotenuse's length is equal to the square root of the sum of the squared coordinates of the point, or $\sqrt{(a^2 + b^2)}$, which is called the *modulus* of the complex number $a + bi$. You denote the modulus as $|a + bi|$ and can use it as the length of a complex number.

In a quantum computer, registers consist of one or more qubits in a state of superposition, which can be characterized by a set of such complex

numbers, or amplitudes. But as you'll see, these amplitudes can't be just any numbers.

Amplitudes of a Single Qubit

You can characterize a single qubit by two amplitudes that we'll denote as α (alpha) and β (beta). You can then express a qubit's state as $\alpha|0\rangle + \beta|1\rangle$, where the $|\ \rangle$ notation denotes vectors in a quantum state. This notation means that when you observe this qubit, it appears as 0 with a probability $|\alpha|^2$ and 1 with a probability $|\beta|^2$. For these to be actual probabilities, $|\alpha|^2$ and $|\beta|^2$ must be numbers between 0 and 1, and $|\alpha|^2 + |\beta|^2$ must be equal to 1.

For example, say you have the qubit Ψ (psi) with amplitudes $\alpha = 1/\sqrt{2}$ and $\beta = 1/\sqrt{2}$. You express this as follows:

$$\Psi = \left(1 / \sqrt{2}\right)|0\rangle + \left(1 / \sqrt{2}\right)|1\rangle = \left(|0\rangle + |1\rangle\right) / \sqrt{2}$$

In the qubit Ψ, the value 0 has an amplitude of $1/\sqrt{2}$, and the value 1 has the same amplitude, $1/\sqrt{2}$. To get the actual probability from the amplitudes, compute the modulus of $1/\sqrt{2}$ (which is equal to $1/\sqrt{2}$ because it has no imaginary part) and then square it: $(1/\sqrt{2})^2 = 1/2$. This means that if you observe the qubit Ψ, you'll have a $1/2$ chance of seeing a 0 and the same chance of seeing a 1.

Now consider the qubit Φ (phi), where:

$$\Phi = \left(i / \sqrt{2}\right)|0\rangle - \left(1 / \sqrt{2}\right)|1\rangle = \left(i|0\rangle - |1\rangle\right) / \sqrt{2}, \text{ or } |\Phi\rangle = \left(i / \sqrt{2},\ -1 / \sqrt{2}\right)$$

The qubit Φ is fundamentally distinct from Ψ because unlike Ψ, where amplitudes have equal values, the qubit Φ has distinct amplitudes of $\alpha = i/\sqrt{2}$ (a positive imaginary number) and $\beta = -1/\sqrt{2}$ (a negative real number). If, however, you observe Φ, the chance of seeing a 0 or 1 is $1/2$, the same as it is with Ψ. Compute the probability of seeing a 0 as follows, based on the preceding rules:

$$|\alpha|^2 = \left(\sqrt{\left(1 / \sqrt{2}\right)^2}\right)^2 = 1 / (\sqrt{2})^2 = 1/2$$

Note that because $\alpha = i/\sqrt{2}$, you can write α as $a + bi$ with $a = 0$ and $b = 1/\sqrt{2}$, and computing $|\alpha| = \sqrt{(a^2 + b^2)}$ yields $1/\sqrt{2}$.

Different qubits can behave similarly to an observer (with the same probability of seeing a 0 for both qubits) but have different amplitudes. This says that the actual probabilities of seeing a 0 or a 1 characterize a qubit only partially; this is similar to observing the shadow of an object on a wall, which provides an idea of the object's width and height but not of its depth. In the case of qubits, this hidden dimension is the value of its amplitude: Is it positive or negative? Is it a real or an imaginary number?

NOTE *To simplify notations, we often write a qubit as its pair of amplitudes (α, β). We can thus write the previous example as $|\Psi\rangle = (1/\sqrt{2},\ 1/\sqrt{2})$.*

Amplitudes of Groups of Qubits

How do we understand multiple qubits? For example, eight qubits can form a *quantum byte* when the quantum states of these eight qubits are connected via entanglement. You can describe such a quantum byte as follows, where the αs are the amplitudes associated with each of the 256 possible values of the group of eight qubits:

$$\alpha_0 |00000000\rangle + \alpha_1 |00000001\rangle + \alpha_2 |00000010\rangle + \alpha_3 |00000011\rangle + \ldots + \alpha_{255} |11111111\rangle$$

Note that you must have $|\alpha_0|^2 + |\langle\alpha_1|^2 + \ldots + |\alpha_{255}|^2 = 1$ so that all probabilities sum to 1.

You can view this group of eight qubits as a set of $2^8 = 256$ amplitudes because it has 256 possible configurations, each with its own amplitude. In physical reality, however, you'd have eight physical objects, not 256. The 256 amplitudes are an implicit characteristic of the group of eight qubits; each of these 256 numbers can take any of infinitely many different values. Generalizing, you can characterize a group of n qubits by a set of 2^n complex numbers, a number that grows exponentially with the numbers of qubits.

If you want to simulate the evolution of a quantum state using a classical computer, you need to store this exponential number of amplitudes and perform calculations to modify them. This requirement is one of the main reasons why a classical computer can't efficiently simulate a quantum computer: doing so requires a gigantic amount of memory (of the order of 2^n) to store the same amount of information contained in just n qubits using a quantum system. In practice, you can simulate a maximum of 50 or 60 qubits, depending on the type of calculation.

Quantum Gates

The concepts of amplitude and *quantum gates* are unique to quantum computing. A quantum gate is essentially a transformation of one or more qubits, and it's the counterpart of electronic gates in the quantum computing realm. Whereas a classical computer uses registers, memory, and a microprocessor to perform a sequence of instructions on data, a quantum computer transforms a group of qubits reversibly by applying a series of quantum gates and then measures the value of one or more qubits. Quantum computers promise more computing power because with only n qubits they can affect the values of 2^n amplitudes. This property has profound implications.

From a mathematical standpoint, quantum algorithms are essentially a circuit of quantum gates that transforms a set of complex numbers (the amplitudes) before a final measurement, where the value of 1 or more qubits is observed (see Figure 14-3).

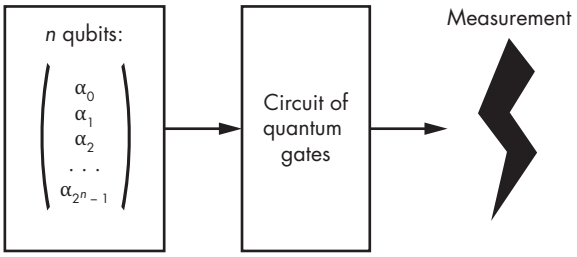


Figure 14-3: Principle of a quantum algorithm

We also refer to quantum algorithms as *quantum gate arrays* or *quantum circuits*.

Quantum Gates as Matrix Multiplications

Unlike the Boolean gates of a classical computer (AND, XOR, and so on), a quantum gate acts on a group of amplitudes just as a matrix acts when multiplied with a vector. For example, to apply the simplest quantum gate, the *identity* gate, to the qubit Φ, we see I as a 2×2 identity matrix and multiply it with the column vector consisting of the two amplitudes of Φ:

$$I|\Phi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} i/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1\times\frac{i}{\sqrt{2}} + 0\times\left(-\frac{1}{\sqrt{2}}\right) \\ 0\times\frac{i}{\sqrt{2}} + 1\times\left(-\frac{1}{\sqrt{2}}\right) \end{pmatrix} = \begin{pmatrix} i/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \Phi\rangle$$

The result of this matrix–vector multiplication is another column vector with two elements, where the top value is equal to the dot product of the I matrix's first line with the input vector (the result of adding the product of the first elements 1 and $i/\sqrt{2}$ to the product of the second elements 0 and $-1/\sqrt{2}$) and likewise for the bottom value.

The identity gate I is pretty useless because it doesn't do anything and leaves a qubit unchanged.

NOTE *In practice, a quantum computer wouldn't explicitly compute matrix–vector multiplications because the matrices would be way too large. (That's why a classical computer can't simulate quantum computing.) Instead, a quantum computer would transform qubits as physical particles through physical transformations that are equivalent to a matrix multiplication. Confused? Here's what Richard Feynman had to say: "If you are not completely confused by quantum mechanics, you do not understand it."*

The Hadamard Quantum Gate

One of the most useful quantum gates is the *Hadamard gate*, usually denoted as H. You can define the Hadamard gate as follows (note the negative value in the bottom-right position):

$$H = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

Applying this gate to the qubit $|\Psi\rangle = (1/\sqrt{2}, 1/\sqrt{2})$ results in the following:

$$H|\Psi\rangle = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 + 1/2 \\ 1/2 - 1/2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

By applying the Hadamard gate H to $|\Psi\rangle$, you obtain the qubit $|0\rangle$ for which the value $|0\rangle$ has amplitude 1, and $|1\rangle$ has amplitude 0. This tells you that the qubit behaves deterministically: if you observe this qubit, you'll always see a 0 and never a 1. In other words, you've lost the randomness of the initial qubit $|\Psi\rangle$.

Applying the Hadamard gate again to the qubit $|0\rangle$ results in the following:

$$H|0\rangle = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = |\Psi\rangle$$

This brings you back to the qubit $|\Psi\rangle$ and a randomized state. We often use the Hadamard gate in quantum algorithms to go from a deterministic state to a uniformly random one.

Not All Matrices Are Quantum Gates

Although you can see the application of quantum gates as matrix multiplications, not all matrices correspond to quantum gates. Recall that a qubit consists of the complex numbers α and β, the amplitudes of the qubit, that satisfy the condition $|\alpha|^2 + |\beta|^2 = 1$. If you get two amplitudes that don't match this condition after multiplying a qubit by a matrix, the result can't be a qubit. Quantum gates correspond only to *unitary matrices*, which preserve the property $|\alpha|^2 + |\beta|^2 = 1$.

Unitary matrices (and quantum gates by definition) are *invertible*, meaning that given the result of an operation, you can compute back the original qubit by applying the *inverse* matrix. This is why quantum computing is a kind of *reversible computing*.

Quantum Speedup

A *quantum speedup* occurs when a quantum computer can solve a problem fundamentally faster than a classical one. For example, to search for an item among n items of an unordered list on a classical computer, you need on average $n/2$ operations because you need to look at each item in the list before finding the one you're looking for. (On average, you'll find that item after searching half of the list.) No classical algorithm can do better than $n/2$. However, a quantum algorithm exists to search for an item in

only $O(\sqrt{n})$ operations, which is orders of magnitude smaller than $n/2$. For example, if n is equal to 1,000,000, then $n/2$ is 500,000, whereas $\sqrt{n}$ is 1,000.

We quantify the difference between quantum and classical algorithms in terms of *time complexity*, which we represent with the $O()$ notation. In the previous example, the quantum algorithm runs in time $O(\sqrt{n})$, but the classical algorithm can't be faster than $O(n)$. Because this difference in time complexity is due to the square exponent, we call this *quadratic speedup*. While such a speedup likely makes a difference, there are much more powerful ones.

Exponential Speedup and Simon's Problem

Exponential speedups are the holy grail of quantum computing. They occur when a task that takes an exponential amount of time on a classical computer, such as $O(2^n)$, can be performed on a quantum computer with polynomial complexity—namely, $O(n^k)$ for some fixed number k. This exponential speedup can turn a practically impossible task into a possible one. (Recall from Chapter 9 that cryptographers and complexity theorists associate exponential time with the impossible and polynomial time with the practical.)

The poster child of exponential speedups is *Simon's problem*. In this computational problem, a function, $\mathbf{f}()$, transforms n-bit strings to n-bit strings, such that the output of $\mathbf{f}()$ looks like a random n-bit string, but with one constraint: there's a secret value, m, such that for any two values x, y, we have $\mathbf{f}(x) = \mathbf{f}(y)$ if and only if $y = x \oplus m$. Simon's problem consists in finding m given black-box access to $\mathbf{f}()$.

Solving Simon's problem with a classical algorithm boils down to finding a collision, or values x and y such that $\mathbf{f}(x) = \mathbf{f}(y)$. This takes approximately $2^{n/2}$ queries to $\mathbf{f}()$. However, Figure 14-4 shows that a quantum algorithm can solve Simon's problem in only approximately n queries, with the extremely efficient time complexity of $O(n)$.

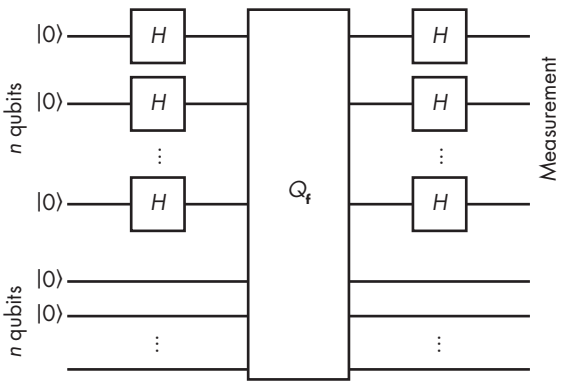


Figure 14-4: The circuit of the quantum algorithm that solves Simon's problem efficiently

In the quantum circuit solving Simon's problem, you initialize $2n$ qubits to $|0\rangle$, apply Hadamard gates (H) to the first n qubits, and then apply the gate $Q_{\mathbf{f}}$ to the two groups of all n qubits. Given two n-qubit groups x and y, the gate $Q_{\mathbf{f}}$ transforms the quantum state $|x\rangle|y\rangle$ to the state $|x\rangle|\mathbf{f}(x) \oplus y\rangle$. That is, it computes the function $\mathbf{f}()$ on the quantum state reversibly because you can go from the new state to the old one by computing $\mathbf{f}(x)$ and XORing it to $\mathbf{f}(x) \oplus y$. (Explaining why this works is beyond this book's scope.)

You can use the exponential speedup for Simon's problem against symmetric ciphers only in very specific cases, but the next section will discuss some real crypto-killer applications of quantum computing.

The Threat of Shor's Algorithm

In 1995, AT&T researcher Peter Shor published an eye-opening article titled "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." *Shor's algorithm* is a quantum algorithm that causes an exponential speedup when solving the factoring, discrete logarithm (DLP), and elliptic curve discrete logarithm (ECDLP) problems. You can't efficiently solve these problems with a classical computer, but you could with a quantum computer. This means that a quantum computer could solve any cryptographic algorithm that relies on those problems, including RSA, Diffie–Hellman, elliptic curve cryptography, and most currently deployed public-key cryptography mechanisms (except for those that transitioned to post-quantum cryptography). In other words, you could reduce the security of RSA or elliptic curve cryptography to that of Caesar's cipher. (Shor might as well have titled his article "Breaking All Public-Key Crypto on a Quantum Computer.") Renowned complexity theorist Scott Aaronson called Shor's algorithm "one of the major scientific achievements of the late 20th century."

Shor's algorithm actually solves a more general class of problems than factoring and discrete logarithms. Specifically, if a function $\mathbf{f}()$ is *periodic*—that is, if there's an ω (the period) such that $\mathbf{f}(x + \omega) = \mathbf{f}(x)$ for any x—then Shor's algorithm will efficiently find ω. (This looks very similar to Simon's problem, as it was a major inspiration for Shor's algorithm.)

Discussing the details of how Shor's algorithm achieves its speedup is far too technical for this book, but the next section shows how to use Shor's algorithm to solve the factoring and discrete logarithm problems (see Chapter 9), the hard problems behind RSA and Diffie–Hellman.

The Factoring Problem

Say you want to factor a large number, $N = pq$. It's easy to factor N if you can compute the period of $a^x \bmod N$, for some constant a. This task is hard to do with a classical computer but easy with a quantum one. First pick a random number a less than N, and ask Shor's algorithm to find the period ω of the function $\mathbf{f}(x) = a^x \bmod N$. Once you've found the period, you'll have $a^x \bmod N = a^{x+\omega} \bmod N$ (that is, $a^x \bmod N = a^x a^\omega \bmod N$), which means

that $a^{\omega} \bmod N = 1$, or equivalently $a^{\omega} - 1 \bmod N = 0$. In other words, $a^{\omega} - 1$ is a multiple of N, meaning that $a^{\omega} - 1 = kN$ for some unknown number k.

When ω is even, it's easy to factor $a^{\omega} - 1$ as $(a^{\omega/2} - 1)(a^{\omega/2} + 1)$, where the value $a^{\omega/2}$ is a *root of unity*, as $(a^{\omega/2})^2 \bmod N = 1$. When the period ω is odd, rerun Shor with another value of a until you get an even number.

As the factors of $a^{\omega} - 1$ contain the prime factors of k and N, you can find these factors distributed among those of $a^{\omega/2} - 1$ and $a^{\omega/2} + 1$. You can then calculate the greatest common divisor (GCD) between $a^{\omega/2} - 1$ and N, and between $a^{\omega/2} + 1$ and N, to obtain a nontrivial factor of N—that is, a value other than 1 or N. If this isn't the case—for example, when $a^{\omega/2} - 1$ or $a^{\omega/2} + 1$ is a multiple of N—restart the attack with another a.

Having obtained the factors of N, you've now recovered the RSA private key from its public key, enabling you to decrypt encrypted messages or forge signatures.

Note that the best classical algorithm to use to factor a number N runs in time exponential in n, the bit length of N (that is, $n = \log_2 N$). However, Shor's algorithm runs in time *polynomial* in n—namely, $O(n^2(\log n)(\log\log n))$. This means that if you had a quantum computer, you could run Shor's algorithm and see the result within a more reasonable amount of time than thousands of years.

The Discrete Logarithm Problem

The challenge in the discrete logarithm problem is to find x, given $y = g^x \bmod p$, for some known numbers g and p. Solving this problem takes a (sub)exponential amount of time on a classical computer, but you can find x easily with Shor's algorithm thanks to its efficient period-finding technique.

For example, consider the function $\mathbf{f}(a, b) = g^a y^b$. Say you want to find the period of this function, the numbers ω and ω', such that $\mathbf{f}(a + \omega, b + \omega') = \mathbf{f}(a, b)$ for any a and b. The solution you seek is then $x = -\omega/\omega'$ modulo q, the order of g, which is a known parameter. The equality $\mathbf{f}(a + \omega, b + \omega') = \mathbf{f}(a, b)$ implies $g^{\omega} y^{\omega'} \bmod p = 1$. By substituting y with g^x, you have $g^{\omega + x\omega'} \bmod p = 1$, which is equivalent to $\omega + x\omega' \bmod q = 0$, from which you derive $x = -\omega/\omega'$.

Again, the overall complexity is $O(n^2(\log n)(\log\log n))$, with n the bit length of p. This algorithm generalizes to find discrete logarithms in any finite commutative group, not just the group of numbers modulo a prime number. You can thus apply it to solve ECDLP as well, the elliptic curve version of the discrete logarithm problem.

Grover's Algorithm

Another important form of quantum speedup is the ability to search among n items in time proportional to the square root of n, whereas any classical algorithm would take time proportional to n. This quadratic speedup is possible thanks to *Grover's algorithm*, a quantum algorithm discovered in 1996. I won't cover the internals of Grover's algorithm because they're

essentially a bunch of Hadamard gates, but I'll explain what kind of problem Grover solves and its potential impact on cryptographic security. I'll also show why you can salvage a symmetric crypto algorithm from quantum computers by doubling the key or hash value size, whereas asymmetric algorithms are destroyed for good.

Think of Grover's algorithm as a way to find the value x among n possible values, such that $\mathbf{f}(x) = 1$, and where $\mathbf{f}(x) = 0$ for most other values. If m values of x satisfy $\mathbf{f}(x) = 1$, Grover will find a solution in time $O(\sqrt{(n/m)})$; that is, in time proportional to the square root of n divided by m. In comparison, a classical algorithm can't do better than $O(n/m)$.

Now consider the fact that $\mathbf{f}()$ can be any function. It could be, for example, "$\mathbf{f}(x) = 1$ if and only if x is equal to the unknown secret key K such that $\mathbf{E}(K, P) = C$" for some known plaintext P and ciphertext C, and where $\mathbf{E}()$ is some encryption function. In practice, this means that if you're looking for a 128-bit AES key with a quantum computer, you'll find the key in time proportional to 2^{64}, rather than 2^{128} if you had only classical computers. You'd need a large enough plaintext to ensure the uniqueness of the key. (If the plaintext and ciphertext are, say, 32 bits, many candidate keys would map that plaintext to that ciphertext.) The complexity 2^{64} is much smaller than 2^{128}, meaning that a secret key would be much easier to recover. But there's an easy solution: to restore 128-bit security, just use 256-bit keys! Grover's algorithm will then reduce the complexity of searching a key to $2^{256 / 2} = 2^{128}$ operations.

Grover's algorithm can also find preimages of hash functions (see Chapter 6). To find a preimage of some value h, we define the $\mathbf{f}()$ function as "$\mathbf{f}(x) = 1$ if and only if $\mathbf{Hash}(x) = h$, otherwise $\mathbf{f}(x) = 0$." Grover thus gets you preimages of n-bit hashes at the cost of the order of $2^{n/2}$ operations. As with encryption, to ensure 2^n post-quantum security, use hash values twice as large, since Grover's algorithm finds a preimage of a $2n$-bit value in at least 2^n operations.

The bottom line is that you can salvage symmetric crypto algorithms from quantum computers by doubling the key or hash value size.

NOTE *There's a famous quantum algorithm that finds hash function collisions in time* $O(2^{n/3})$*, instead of* $O(2^{n/2})$*, as with the classic birthday attack. This suggests that quantum computers can outperform classical computers for finding hash function collisions, except that the* $O(2^{n/3})$*-time quantum algorithm also requires* $O(2^{n/3})$ *space, or memory, to run. Give* $O(2^{n/3})$*'s worth of computer space to a classic algorithm, and it can run a parallel collision search algorithm with a collision time of only* $O(2^{n/6})$*, which is much faster than the* $O(2^{n/3})$ *quantum algorithm. (For details of this attack, see "Cost Analysis of Hash Collisions" by Daniel J. Bernstein at* https://cr.yp.to/papers.html#collisioncost*.) In 2017, however, cryptographers proposed a quantum algorithm finding collisions in time* $O(2^{2n/5})$*, requiring* $O(n)$ *quantum memory and* $O(2^{n/5})$ *classical memory. This may outperform classical search (see* https://eprint.iacr.org/2017/847*).*

Why Is It So Hard to Build a Quantum Computer?

Although quantum computers can in principle be built, we don't know how hard it will be or when that might happen, if at all. As of mid-2024, the record holder is a machine with 1,121 qubits (IBM's "Condor"), whereas we'd need to keep millions of qubits stable for weeks to break any crypto. The point is, we're not there yet.

The difficulty of building a quantum computer stems from needing extremely small things to play the role of qubits—smaller than atoms, such as photons. Because qubits must be so small, they're also extremely fragile.

Also, qubits must be kept at extremely low temperatures (close to absolute zero) to remain stable. Even at freezing temperatures, the state of qubits decays, and they eventually become useless. As of this writing, we don't yet know how to make qubits that last for more than a couple of seconds (their coherence time).

Another challenge is that the environment, such as heat and magnetic fields, can affect the qubits' states and lead to computation errors. In theory, it's possible to correct these errors, but it's difficult to do so. Correcting qubits' errors requires quantum error-correcting codes, which in turn require many additional qubits and a low enough rate of error.

At the moment, there are two main approaches to forming qubits: superconducting circuits and ion traps. Labs at Google and IBM champion using *superconducting circuits*, which is based on forming qubits as tiny electrical circuits that rely on quantum phenomena from superconductor materials, where charge carriers are pairs of electrons. Qubits made of superconducting circuits have a very short lifetime.

Ion traps, or trapped ions, consist of ions (charged atoms) and are manipulated using lasers to prepare the qubits in specific initial states. Ion traps tend to be more stable than superconducting circuits, but they're slower to operate and seem harder to scale.

Building a quantum computer is really a moon shot effort. The challenge comes down to 1) building a system with a handful of qubits that's stable, fault tolerant, and capable of applying basic quantum gates, and 2) scaling such a system to thousands or millions of qubits to make it useful. From a purely physical standpoint and to the best of our knowledge, there's nothing to prevent the creation of large fault-tolerant quantum computers. But many things are possible in theory and prove hard or too costly to realize in practice (like secure computers). The future will tell who is right—the quantum optimists (who predict a large quantum computer within the decade) or the quantum skeptics (who argue that the human race will never see a quantum computer).

As mentioned earlier, as of January 2024, one of the most advanced achievements is IBM's quantum computing chip Condor that includes 1,121 qubits, a technology based on superconducting circuits. But the number of qubits shouldn't be the only metric when comparing quantum computing systems. Other important factors include the stability time, the number of qubits entangled together, and the ability to reliably correct errors.

Post-Quantum Cryptographic Algorithms

The field of *post-quantum cryptography* focuses on designing public-key algorithms that a quantum computer can't break; that is, they're quantum safe and can replace RSA and elliptic curve–based algorithms in a future where off-the-shelf quantum computers could break 4,096-bit RSA moduli in a snap.

Such algorithms shouldn't rely on a hard problem known to be efficiently solvable by Shor's algorithm, which kills the hardness in factoring and discrete logarithm problems. Symmetric algorithms such as block ciphers and hash functions would lose only half their theoretical security in the face of a quantum computer but wouldn't be as badly broken as RSA. They might constitute the basis for a post-quantum scheme.

In the following sections, we'll review the four main types of post-quantum algorithms: code based, lattice based, multivariate, and hash based.

Code-Based Cryptography

Code-based post-quantum cryptographic algorithms are based on *error-correcting codes*, which are techniques designed to transmit bits over a noisy channel. The basic theory of error-correcting codes dates back to the 1950s. The first code-based encryption scheme (the *McEliece* cryptosystem) was developed in 1978 and is still unbroken. You can use code-based crypto schemes for both encryption and signatures. Their main limitation is the size of their public key, which is typically on the order of a hundred kilobytes. But is that really a problem when the average size of a web page is around 2MB?

Let me first explain what error-correcting codes are. Say you want to transmit a sequence of bits as a sequence of 3-bit words, but the transmission's unreliable and you're concerned about incorrectly transmitting one or more bits: you send 010, but the receiver gets 011. One might address this by using a basic error-correction *repetition code*: instead of transmitting 010, you transmit 000111000 (repeating each bit three times), and the receiver decodes the received word by taking the majority value for each of the 3 bits.

For example, a receiver would decode the repetition codeword 100110111 to 011 because 100 contains two 0s, then 110 contains two 1s, and 111 contains three 1s. This particular error-correcting code allows a receiver to correct only up to one error per 3-bit chunk, because if two errors occur in the same 3-bit chunk, the majority value would be the wrong one.

Linear codes are a less trivial example of error-correcting codes. In the case of linear codes, a word to encode is seen as an n-bit vector v, and encoding consists of multiplying v with an $m \times n$ matrix G to compute the code word $w = vG$. (In this example, m is greater than n, meaning the code word is longer than the original word.) One can structure the matrix G such that for a given number t, any t-bit error in w allows the recipient to recover the correct v. In other words, t is the maximum number of errors that one can correct.

To encrypt data using linear codes, the McEliece cryptosystem constructs G as a secret combination of three matrices and encrypts by computing $w = vG$ added with some random value, e, which has a fixed number of bits set to 1. Here, G is the public key, and the private key is composed of the matrices A, B, and C such that $G = ABC$. Knowing A, B, and C allows one to decode a message reliably and retrieve w. But without these matrices, it should be impossible to decode the word and thus to decrypt.

The security of the McEliece encryption scheme relies on the hardness of decoding a linear code with insufficient information, a problem we know to be **NP**-hard and therefore out of reach of quantum computers. Bear in mind, however, that just because a problem is **NP**-hard doesn't mean that all its instances will be impossible to solve in practice. It's therefore necessary to evaluate which instances of the difficult problem are presented by a cryptosystem and whether these instances will always be difficult. McEliece's cipher satisfies this criterion after years of analysis by cryptographers and coding theory experts.

Lattice-Based Cryptography

Lattices are mathematical structures that essentially consist of a set of points in an n-dimensional space, with some periodic structure. For example, Figure 14-5 shows how you can view a lattice in dimension two ($n = 2$) as the set of points.

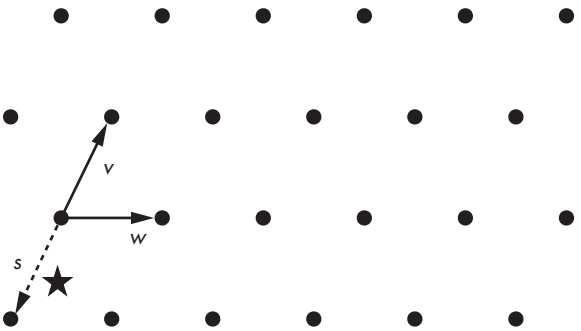


Figure 14-5: Points of a two-dimensional lattice, where v *and* w *are basis vectors of the lattice and* s *is the closest vector to the star-shaped point*

Lattice theory has led to deceptively simple cryptography schemes. I'll give you the gist of it.

Short integer solution (SIS) is a hard problem in lattice-based crypto that consists of finding the secret vector s of n numbers given (A, b) such that $b = As \bmod q$, where A is a random $m \times n$ matrix and q is a prime number.

Another hard problem in lattice-based cryptography, *learning with errors (LWE)*, consists of finding the secret vector s of n numbers given (A, b), where $b = As + e \bmod q$, with A being a random $m \times n$ matrix, e a random vector of noise, and q a prime number. This problem looks a lot like noisy decoding in code-based cryptography.

SIS and LWE are somewhat equivalent, and we can restate them as instances of the *closest vector problem (CVP)* on a lattice, or the problem of finding the vector in a lattice closest to a given point, by combining a set of basis vectors. The dotted vector s in Figure 14-5 shows how to find the closest vector to the star-shaped point by combining the basis vectors v and w.

CVP and other lattice problems are believed to be hard both for classical and quantum computers. But this doesn't directly transfer to secure cryptosystems, because some problems are hard only in the worst case (that is, for their hardest instance) rather than the average case (which is what we need for crypto). Furthermore, while finding the exact solution to CVP is hard, finding an approximation of the solution can be considerably easier.

That said, post-quantum cryptosystems based on lattices have proven to offer the best combination of security and performance. The standards chosen by NIST in 2022 are mainly from this family, as you'll see later.

Multivariate Cryptography

Multivariate cryptography focuses on building cryptographic schemes that are as hard to break as it is to solve systems of multivariate equations, or equations involving multiple unknowns that are multiplied together in the equations. Consider, for example, the following system of equations involving four unknowns x_1, x_2, x_3, x_4:

$$\begin{aligned} x_1x_2 + x_3x_4 + x_2 &= 1 \\ x_1x_3 + x_1x_4 + x_2x_3 &= 12 \\ {x_1}^2 + {x_3}^2 + x_4 &= 4 \\ x_2x_3 + x_2x_4 + x_1 + x_4 &= 0 \end{aligned}$$

These equations consist of the sum of terms that are either a single unknown, such as x_4 (or terms of degree one), or the product of two unknown values, such as x_2x_3 (terms of degree two or *quadratic* terms). To solve this system, you need to find the values of x_1, x_2, x_3, x_4 that satisfy all four equations. Equations may be over all real numbers, integers only, or over finite sets of numbers. In cryptography, however, equations are typically over numbers modulo some prime numbers or over binary values (0 and 1).

The hard problem here is finding a solution to a random quadratic system, which is known as *multivariate quadratics (MQ)*. This problem is **NP**-hard and is therefore a potential basis for post-quantum systems because quantum computers won't solve **NP**-hard problems efficiently.

Unfortunately, building a cryptosystem on top of MQ isn't so straightforward. For example, if you were to use MQ for signatures, the private key might consist of three systems of equations: L_1, N, and L_2. Combining them in this order results in another system of equations called P, the public key. Applying the transformations L_1, N, and L_2 consecutively (that is, transforming a group of values as per the system of equations) is then equivalent

to applying P by transforming x_1, x_2, x_3, x_4 to y_1, y_2, y_3, y_4, for example, defined as follows:

$$\begin{aligned} y_1 &= x_1x_2 + x_3x_4 + x_2 \\ y_2 &= x_1x_3 + x_1x_4 + x_2x_3 \\ y_3 &= {x_1}^2 + {x_3}^2 + x_4 \\ y_4 &= x_2x_3 + x_2x_4 + x_1 + x_4 \end{aligned}$$

In such a cryptosystem, L_1, N, and L_2 are chosen such that L_1 and L_2 are linear transformations (that is, having equations where terms are only added, not multiplied) that are invertible, and where N is a quadratic system of equations that is also invertible. This makes the combination of the three an invertible quadratic system, but its inverse is hard to determine without knowing the inverses of L_1, N, and L_2.

Computing a signature then consists of computing the inverses of L_1, N, and L_2 applied to some message, M, which we see as a sequence of variables, x_1, x_2, . . . :

$$S = {L_2}^{-1}\left(N^{-1}\left({L_1}^{-1}(M)\right)\right)$$

Verifying a signature then consists of verifying that $P(S) = M$.

Attackers could break such a cryptosystem if they manage to compute the inverse of P or to determine L_1, N, and L_2 from P. The actual hardness of solving such problems depends on the parameters of the scheme, such as the number of equations used and the size and type of the numbers. But choosing secure parameters is hard, and more than one "safe" multivariate scheme has been broken.

Multivariate cryptography isn't used in major applications due to the challenge of achieving a reliable trade-off between security and performance. A practical benefit of multivariate signature schemes, however, is that they produce short signatures.

Hash-Based Cryptography

Unlike the previous schemes, hash-based cryptography is based on the well-established security of cryptographic hash functions rather than on the hardness of mathematical problems. Because quantum computers can't break hash functions, they can't break anything that relies on the difficulty of finding collisions or preimages, which is the key idea of hash function–based signature schemes.

Hash-based cryptographic schemes are pretty complex, so we'll take a look at their simplest building block: the *Winternitz one-time signature (WOTS)*, a trick discovered around 1979. Here *one-time* means that you can use a private key to sign only one message; otherwise, the signature scheme becomes insecure. (You can combine WOTS with other methods to sign multiple messages, as you'll see shortly.)

Say you want to sign a message viewed as a number between 0 and $w - 1$, where w is some parameter of the scheme. The private key is a random string,

K. To sign a message, M, with $0 \leq M < w$, compute **Hash**(**Hash**(. . .(**Hash**(K))), where the hash function **Hash** repeats M times. You denote this value as $\mathbf{Hash}^M(K)$. The public key is $\mathbf{Hash}^w(K)$, or the result of w nested iterations of **Hash**, starting from K.

A WOTS signature is verified, S, by checking that $\mathbf{Hash}^{w-M}(S)$ is equal to the public key $\mathbf{Hash}^w(K)$. Note that S is K after M applications of **Hash**, so if you do another $w - M$ applications of **Hash**, you'll get a value equal to K hashed $M + (w - M) = w$ times, which is the public key.

This scheme has significant limitations:

Attackers can forge signatures From $\mathbf{Hash}^M(K)$, the signature of M, you can compute $\mathbf{Hash}(\mathbf{Hash}^M(K)) = \mathbf{Hash}^{M+1}(K)$, which is a valid signature of the message $M + 1$. You can fix this problem by signing not only M but also $w - M$, using a second key.

It works for only short messages If messages are 8 bits long, there are up to $2^8 - 1 = 255$ possible messages, so you have to compute **Hash** up to 255 times to create a signature. That might work for short messages, but not for longer ones—for example, with 128-bit messages, signing the message $2^{128} - 1$ takes forever. A workaround is to split longer messages into shorter ones and sign each chunk independently.

It works only once If you use a private key to sign more than one message, an attacker can recover enough information to forge a signature. For example, if $w = 8$ and you sign the numbers 1 and 7 using the preceding trick to avoid trivial forgeries, the attacker gets $\mathbf{Hash}^1(K)$ and $\mathbf{Hash}^7(K')$ as a signature of 1, and $\mathbf{Hash}^7(K)$ and $\mathbf{Hash}^1(K')$ as a signature of 7. From these values, the attacker can compute $\mathbf{Hash}^x(K)$ and $\mathbf{Hash}^x(K')$ for any x in [1;7] and thus forge a signature on behalf of the owner of K and K'. There's no simple way to fix this.

State-of-the-art hash-based schemes rely on more complex versions of WOTS, combined with tree data structures and sophisticated techniques designed to sign different messages with different keys. Unfortunately, the resulting schemes produce large signatures (on the order of dozens of kilobytes, as with SPHINCS+, one of the signature algorithms chosen for standardization by NIST in 2022).

You should also note the difference between *stateful* and *stateless* signature schemes. SPHINCS+ is stateless, whereas XMSS is stateful, as it needs to maintain a counter. Statefulness greatly simplifies algorithm design but forces users to maintain a state such as a counter while using the algorithm.

Finally, note that public-key constructions relying on only hash functions can offer signature schemes but not encryption.

The NIST Standards

In 2017, NIST organized an open competition to identify suitable post-quantum algorithm standards for encryption and signature. Like the previous competitions that gave us AES (Rijndael) and SHA-3 (Keccak), NIST's Post-Quantum Cryptography Standardization project invited

cryptographers to submit algorithms and cryptanalyze other submitters' algorithms to eliminate them from the competition.

NIST received 69 submissions, most of which were lattice based. Of these submissions, 26 made it to the second round. In July 2020, NIST selected seven finalist and eight alternate algorithms. In July 2022, NIST announced the first four standards:

CRYSTALS-Kyber A lattice-based *key encapsulation mechanism (KEM)*, which is a primitive that can be seen as an encryption scheme for secret keys. It can be used to encrypt data (within a hybrid scheme, where a symmetric cipher actually encrypts the data using a key encrypted by the KEM) and used for key agreement, in a similar way as Diffie–Hellman protocols.

CRYSTAL-Dilithium A lattice-based signature scheme designed by the same team as CRYSTALS-Kyber.

Falcon A lattice-based signature scheme based on slightly different techniques and assumptions than Dilithium.

SPHINCS+ A hash-based signature scheme, thus the only algorithm not based on lattices.

NIST stated the following regarding having two lattice-based signature schemes:

> [Both] were selected for their strong security and excellent performance, and NIST expects them to work well in most applications. Falcon will also be standardized by NIST since there may be use cases for which CRYSTALS-Dilithium signatures are too large.

The shortest Dilithium signatures are approximately 2KB long, whereas Falcon's are half as long. NIST also stated that it will standardize SPHINCS+ "to avoid relying only on the security of lattices for signatures."

At the time of writing, draft standards have been published under the FIPS series for Kyber, Dilithium, and SPHINCS+ as FIPS 203 (Module-Lattice-Based Key-Encapsulation Mechanism Standard), FIPS 204 (Module-Lattice-Based Digital Signature Standard), and FIPS 205 (Stateless Hash-Based Digital Signature Standard), respectively, while Falcon's standard is expected a bit later.

These post-quantum algorithms are expected to first be used in hybrid modes, in combination with a classical, non-quantum-resilient algorithm to hedge the risk of weaknesses. For example, Kyber is generally used in combination with X25519, the Diffie–Hellman scheme relying on Curve25519, to protect TLS connections.

NIST also announced four algorithms advancing to the "fourth round." These include the three code-based encryption schemes BIKE, Classic McEliece, and HQC. The isogeny-based SIKE was found to be completely broken shortly after the announcement and thus withdrew from the competition.

NIST started a new project in summer 2022 to identify more post-quantum signature schemes, stating that "signature schemes that are not

based on structured lattices are of greatest interest" and expecting submissions with "short signatures and fast verification." In June 2023, NIST received 50 submissions, including 11 multivariate, 7 lattice-based, 6 code-based, 2 hash-based, and 7 that use *MPC-in-the-head*. This emerging technique turns a multiparty-computation (MPC) protocol into a zero-knowledge proof of knowledge—that is, a single piece of data whose verification corresponds to the verification of a signature. The proposed number and variety of schemes will lead to new attacks and new attack techniques, and hopefully to reliable standards.

How Things Can Go Wrong

Post-quantum cryptography may be fundamentally stronger than RSA or elliptic curve cryptography, but it's not infallible or omnipotent. Our understanding of the security of post-quantum schemes and their implementations is more limited than otherwise, which brings with it increased risk.

Unclear Security Level

Post-quantum schemes can appear deceptively strong yet prove insecure against both quantum and classical attacks. Lattice-based algorithms, such as the ring-LWE family of computational problems (versions of the LWE problem that work with polynomials), are sometimes problematic. Ring-LWE is attractive for cryptographers because we can leverage it to build cryptosystems that are in principle as hard to break as it is to solve the hardest instances of ring-LWE problems, which can be **NP**-hard. But when security looks too good to be true, it often is.

Security proofs are often asymptotic, meaning they're true only for large parameter values such as the dimension of the underlying lattice. However, in practice much smaller parameters are used. Even when a lattice-based scheme looks to be as hard to break as some **NP**-hard problem, its security remains hard to quantify. In the case of lattice-based algorithms, we rarely have a clear picture of the best attacks against them and the cost of such an attack in terms of computation or hardware, due to a lack of understanding of these recent constructions. This uncertainty makes lattice-based schemes harder to compare against better-understood constructions such as RSA. However, researchers have been making progress on this front and, ideally in a few years, lattice problems will be as well understood as RSA. (For more technical details on the ring-LWE problem, read Peikert's excellent survey at *https://eprint.iacr.org/2016/351*.)

The Eventual Existence of Large Quantum Computers

Imagine this CNN headline circa April 2, 2048: "ACME, Inc. reveals its secretly built quantum computer, launches break-crypto-as-a-service platform." OK, RSA and elliptic curve crypto are screwed. Now what?

The bottom line is that post-quantum encryption is way more critical than post-quantum signatures. Let's look at the case of signatures first. If

you were still using RSA-PSS or ECDSA as a signature scheme, you could issue new signatures using a post-quantum signature scheme to restore your signatures' trust. You'd revoke your older, quantum-unsafe public keys and compute fresh signatures for every message you'd signed. After a bit of work, you'd be fine.

You'd have a reason to panic only if you were encrypting data using quantum-unsafe schemes, such as RSA-OAEP. In this case, all transmitted ciphertext could be compromised, so it would be pointless to reencrypt that plaintext with a post-quantum algorithm since your data's confidentiality is already gone.

But what about key agreement, with Diffie–Hellman (DH) and its elliptic curve counterpart (ECDH)? At first glance, the situation looks to be as bad as with encryption: attackers who've collected public keys g^a and g^b could use their shiny new quantum computer to compute the secret exponent a or b and compute the shared secret g^{ab}, and then derive from it the keys that encrypted your traffic. But in practice, Diffie–Hellman isn't always used in such a simplistic fashion. The actual session keys that encrypt your data may be derived from both the Diffie–Hellman shared secret and some internal state of your system. That's how state-of-the-art mobile messaging systems work, thanks to a protocol pioneered with the Signal application. When you send a new message to a peer with Signal, it computes a new Diffie–Hellman shared secret and combines it with internal secrets that depend on the previous messages sent within that session (which can span long periods of time). Such advanced use of Diffie–Hellman makes the work of an attacker much harder, even with a quantum computer.

Implementation Issues

In practice, post-quantum schemes consist of code—software running on some physical processor—not just abstract algorithms. However strong the algorithms may be on paper, they won't be immune to implementation errors, software bugs, or side-channel attacks. An algorithm may be completely post-quantum in theory but may still be broken once implemented—for example, by a classical computer program because a programmer forgot to enter a semicolon.

Furthermore, schemes such as code-based and lattice-based algorithms rely heavily on mathematical operations, the implementation of which uses a variety of tricks to make those operations as fast as possible. By the same token, the complexity of the code in these algorithms makes implementation more vulnerable to side-channel attacks, such as timing attacks, which infer information about secret values based on measurement of execution times. In fact, we've already applied such attacks to code-based encryption (see *https://eprint.iacr.org/2010/479*) and to lattice-based signature schemes (see *https://eprint.iacr.org/2016/300*).

Ironically, using post-quantum schemes may be less secure in practice at first than non-post-quantum ones due to potential vulnerabilities in their implementations.

Further Reading

To learn the basics of quantum computation, read the classic *Quantum Computation and Quantum Information (Anniversary Edition)* by Michael Nielsen and Isaac Chuang (Cambridge, 2011). Scott Aaronson's *Quantum Computing Since Democritus* (Cambridge, 2013), a less technical and more entertaining read, covers more than quantum computing.

Several software simulators allow you to experiment with quantum computing, such as The Quantum Computing Playground at *https://www.quantumplayground.net* or IBM's platform at *https://quantum.ibm.com*. These sites are relatively easy to use, thanks to intuitive visualizations.

For the latest research in post-quantum cryptography, see *https://pqcrypto.org* and the associated conference PQCrypto.

The coming years promise to be particularly exciting for post-quantum crypto, thanks to the continuation of NIST's Post-Quantum Cryptography Standardization project and the deployment at scale of post-quantum solutions.

15

CRYPTOCURRENCY CRYPTOGRAPHY

This chapter wasn't in the initial edition of this book when it came out in fall 2017, a period when cryptocurrency and blockchain were at peak hype. While blockchain hasn't quite lived up to its promises of disrupting several industries, it has profoundly influenced cryptographic research and engineering. Blockchain applications have brought new exciting problems, attracted fresh talent, and offered a novel way to bridge theory and practice.

Before "crypto" became synonymous with cryptocurrency, cryptographic algorithms and protocols largely pertained to standard functionalities such as encryption and secure channels. The more arcane protocols were restricted to niche research areas and technical articles, typically catering to researchers' interests and presented only in academic

conferences. New algorithms were predominantly crafted by academic researchers and would see real-world deployment, if ever, after at least five years of peer review and analysis and after numerous research papers detailing unsuccessful cryptanalysis attempts.

Blockchain turned that process upside down. Much like how the cryptographic protocols of Signal and Tor sidestepped the conventional academic route, blockchain enthusiasts were less bound by tradition. Groundbreaking protocols often debuted in blog posts or informal white papers, with implementation following shortly after. Occasionally, they'd forgo written specifications altogether, letting the code speak for itself. Only after widespread adoption would academia take notice. The most notable case is the Bitcoin protocol, which didn't undergo formal peer review before being deployed.

Many seasoned researchers identified novel challenges within the blockchain domain, often collaborating directly with blockchain entities. They developed complex protocols that not only pushed boundaries and garnered peer recognition but also saw rapid real-world implementation, impacting thousands, if not millions, of systems. Prime instances include efficient ECDSA threshold signature schemes and zero-knowledge proof systems. In some scenarios, existing protocols with little practical use found impactful use cases, such as Boneh–Lynn–Shacham (BLS) signatures.

This chapter provides an overview of these cryptographic algorithms and protocols—those tailored for blockchain and those that thrived due to it. I won't delve into defining blockchain or its workings, as there are myriad online resources. Instead, I emphasize the cryptographic schemes underpinning blockchains, which hold significance irrespective of the blockchain use cases. Even if you remain skeptical of the blockchain phenomenon, I trust you'll find this chapter enlightening.

Hashing Applications

Hash functions, the Swiss Army knives of cryptography, serve numerous applications in blockchain systems, including:

Hashing a transaction's data

Hash functions can produce a digest signed by the transaction's issuer, typically with ECDSA-secp256k1 or Ed25519. Verifying the signature ensures that the owner of a given address approved of the information hashed, leading to including the transaction in the chain's ledger.

In the blockchain Ethereum, a transaction's hash serves as its unique identifier; for example, you can enter a hash in the search field of *https://etherscan.io* to retrieve the associated transaction. Ethereum transaction hashes use Keccak-256—which is similar but not identical to SHA3-256—and process-encoded data. This data includes the recipient's address, the amount of ether (ticker ETH) sent, any smart contract input, the gas price and limit, a nonce (increasing for each new transaction), and a signature of the transaction.

Hashing the content of a block

Hash functions can include the digest in the subsequent block to "chain" the blocks. For example, each Bitcoin block has a *block header* that includes the previous block's hash, a tree hash of the recorded transactions, and some metadata (version, timestamp, nonce, and so on). Bitcoin computes the previous block's hash by hashing the header of the previous block using double SHA-256, or **SHA-256**(**SHA-256**(block header)). This unorthodox construction eliminates the risk of length extension attacks and adds a safety net in case SHA-256 was found to be insecure—which is unlikely.

Aside from these basic use cases, hash functions are the main building block in key components of blockchain systems.

Merkle Trees

Merkle trees are a type of hash tree, which is a data structure computing a root value from leaf values according to a tree pattern. In hash trees, the parent node is computed by hashing the child nodes. Merkle trees are, for example, used to create a Bitcoin block's header. They're named after computer scientist Ralph Merkle, whose 1979 cryptographic scheme construction used binary hash trees.

Tree Hashing Computation

A Merkle tree takes as input values that constitute its *leaves*—that is, the values from which the *root* (the output) is computed. You generally represent data structure trees with their root at the top and leaves at the bottom.

For example, Figure 15-1 shows a Merkle tree hashing four values (*A*, *B*, *C*, *D*).

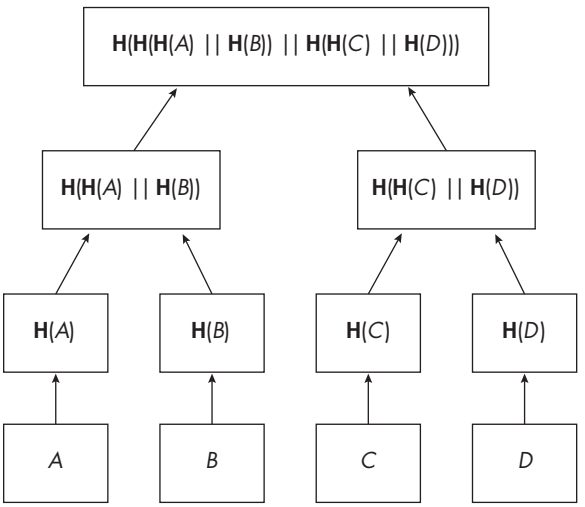


Figure 15-1: A hash tree, where the leaves are the input and the root is the output

The tree hashing then works as follows:

- Hashing each of the four values to obtain the four hashes $\mathbf{H}(A)$, $\mathbf{H}(B)$, $\mathbf{H}(C)$, and $\mathbf{H}(D)$.
- Hashing together each consecutive pair of hashes to obtain $\mathbf{H}(\mathbf{H}(A) \,||\, \mathbf{H}(B))$ and $\mathbf{H}(\mathbf{H}(C) \,||\, \mathbf{H}(D))$. Here you concatenate the hash values (as the || symbol denotes).
- Hashing the two hashes together to obtain the root of the tree, which is the final output of the tree hashing: $\mathbf{H}(\mathbf{H}(\mathbf{H}(A) \,||\, \mathbf{H}(B)) \,||\, \mathbf{H}(\mathbf{H}(C) \,||\, \mathbf{H}(D)))$.

If you omit the initial hashing of the input values—which is necessary only if the values aren't already the size of the hash—you go from four values to one with a tree of two layers, or of *height* two. Note that while the input data may be a different size, all hash values are the same size. Generally, a tree of n layers has 2^n leaves, allowing you to process up to 2^n values by computing $2^n - 1$ hashes from the (hashed) leaves.

If the number of input values isn't exactly 2^n for some integer n, a common technique is to add dummy values (for example, set to zero or to the last value in the list); the padding rule should, however, be carefully chosen to avoid trivial collisions.

Merkle Proofs

A Merkle tree's structure can be leveraged to prove that a given value belongs to the list of 2^n values hashed without recomputing the whole tree (which takes on the order of 2^n operations) but only in time proportional to the *height* of the tree—that is, its number of layers n. Depending on the context, such proofs are called a *membership path*, an *inclusion proof*, or a *Merkle proof*. This is a Merkle tree's killer feature, which general-purpose hash functions don't offer.

Figure 15-2 shows how this works. The shaded cells are the values sufficient to prove that V_1 is one of the values hashed to obtain the root.

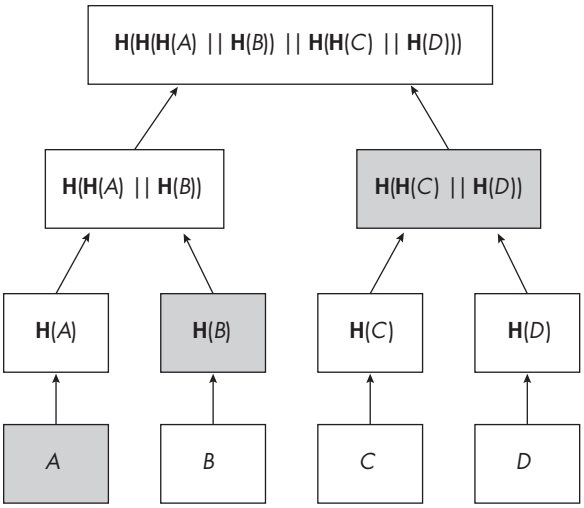


Figure 15-2: A Merkle tree, wherein the shaded cells constitute the membership path of A

Say you want to prove that A was one of the values hashed, without revealing B, C, or D. First, hash A to obtain the actual leaf of the hash tree. Then, assume that you've received A's membership path, which consists of the other shaded values, $\mathbf{H}(B)$ and $\mathbf{H}(\mathbf{H}(C) \parallel \mathbf{H}(D))$. To verify that A belongs to the tree, compute the following:

- $X = \mathbf{H}(\mathbf{H}(A) \parallel \mathbf{H}(B))$, since you know A and $\mathbf{H}(B)$
- $\mathbf{H}(\mathbf{H}(X) \parallel \mathbf{H}(\mathbf{H}(C) \parallel \mathbf{H}(D)))$, since you know $\mathbf{H}(\mathbf{H}(C) \parallel \mathbf{H}(D))$

It took only two hashes of sibling values to prove A's inclusion, or as many hashes as the height of the tree. A tree with eight leaves has a height of three; thus, a membership path verification needs three sibling hashes. With 16 leaves, you need four sibling hashes, and so on, with n sibling hashes for a tree with 2^n leaves.

Blockchain applications often use Merkle trees to hash transactions into a single *Merkle root*, such as that included in a Bitcoin block header. A typical Bitcoin block registers about 2,000 transactions, which requires a tree of height 11 ($2^{11} = 2{,}048$). In this case, it takes 2,048 – 1 = 2,047 sibling hashes to compute the root from the leaves, where each leaf is the hash of a transaction's data. Each hash computes a double SHA-256, so in the case of 2,048 transactions, it takes 2,047 + 2,048 = 4,095 double SHA-256s, or 8,190 calls to SHA-256.

Ethereum uses a slightly more complex tree-based data structure, combining Merkle trees with *Patricia tries* (this is not a typo; "trie" comes from "retrieval"), a tree-like structure that stores key-value pairs. This structure serves Ethereum's state-based model, which significantly differs from Bitcoin's UTXO (unspent transaction output) model, for which simpler Merkle trees are sufficient.

Proof of Work

Proof of work (PoW) is arguably the most critical component of a blockchain's consensus protocol—for those based on PoW rather than proof of stake (PoS) or other protocols.

A *PoW* is essentially a hash function that takes some fixed inputs and variable inputs and whose result must match a certain pattern to be valid. Parties that aim to *solve* a PoW repeatedly compute the hash with different values of the variable inputs until the result satisfies some constraint.

For example, in Bitcoin and some other PoW-based blockchains, the constraint is that the hash value when seen as a 256-bit number must be less than a given number. This can also be seen as the hash value having a given number of leading zeros, when seeing the hash as the big-endian encoding of a 256-bit number.

For example, in 2022 the highest Bitcoin difficulty value (as reported on *https://btc.com/stats/diff*) was 34,244,331,613,176, or approximately 2^{45}. You multiply this by 2^{32} to find the actual value that the hash must be less than, 2^{77}. For each block, all the network participants (*miners*) jointly compute in the order of 2^{77} double SHA-256 computations to find a solution to the PoW. Such a solution consists of a nonce (the variable part of the PoW hash's input) that yields a hash value less than 2^{77}. The fixed part of the PoW hash's input consists of the block header values (version, previous block hash, Merkle tree root, timestamp, and difficulty target value).

Without a PoW "slowing down" the production of blocks in a blockchain, new valid blocks could be produced instantaneously, meaning that histories of transactions could be produced and re-created at will. It would be impossible to achieve *finality* (that is, the assurance that a block, and thus a set of transactions, cannot be reverted or changed once accepted by the network). In particular, it would be impossible to protect the network against *double spending*, or spending the same coins in two distinct transactions.

Not all PoW schemes are as simple as Bitcoin's, which uses a general-purpose hash function (SHA-256). Several PoWs attempted to make its computation on dedicated hardware less efficient compared to general-purpose CPUs to discourage the centralization of mining by organizations investing in the development of optimized hardware miner technology—as opposed to off-the-shelf servers and computers that anyone can use. Implemented tricks include the following:

Memory hardness You can force the PoW to use a large amount of memory, typically by generating a huge table and accessing at unpredictable addresses. For example, Ethereum's PoW used the Ethash algorithm, which required approximately 4GB of memory. (In September 2022, Ethereum abandoned Ethash and switched from a PoW to a proof-of-stake mechanism.)

Virtual machines As some malware do, you can create a custom set of computer instructions that would be translated to standard instructions by a virtual machine application, while also potentially using a large

amount of memory to compute a PoW's solution. This is the approach of RandomX, a PoW algorithm adopted by the Monero blockchain.

Hierarchical Key Derivation

Blockchain users generally want to manage multiple accounts, such that each account consists of a key pair where:

- The *private key* must be secret, as it's the signing key required to sign transactions and transfer funds from the account.
- The *public key* must be public, as it's used to verify signatures of transactions. It's also the value from which the account's address is derived. For example, Bitcoin derives addresses from a public key using a combination of SHA-256 and RIPEMD-160 hashing. Note that in some blockchains, such as Bitcoin, the public key is not public until the account issues a transaction.

To reliably manage all these keys, blockchain developers defined *hierarchical deterministic wallets (HD wallets)*, which make key management less cumbersome and risky than having to generate and back up a fresh random key for each new account.

With HD wallets, you generate and store one secret, the *seed* (also called the *master key* or *entropy*). This seed is the only randomly generated value and the only one providing entropy, or uncertainty, and therefore secrecy to the private signing keys derived from it. Wallet software applications often encode the seed as a *seed phrase*, or *mnemonic*, a sequence of 12 to 24 words that makes it easier to save and memorize.

Let's review how this key derivation works using the HMAC-SHA-512 pseudorandom function (the HMAC construction instantiated with SHA-512): from a seed S of 128 or 256 bits, compute HMAC-SHA-512 with the seed as a key input, and as a message input an identifier of the underlying elliptic curve (a string that can be "Bitcoin seed," "Nist256p1 seed," or "ed25519 seed"). The first 256 bits of the result are the master key, k, and the last 256 bits are the master chain code, c, a value used to derive more keys.

You can derive a *child private key* from k and c as follows, given an identifier i (a number at most 2^{31}): compute HMAC-SHA-512 with the chain code c as a key, and a message including k and i. The 512-bit result is parsed as a 256-bit value L followed by a 256-bit chain code R. In simplified notation, $L \,||\, R = \text{HMAC-SHA-512}(c, k \,||\, i)$. You then set the child private key to $k + L$, and its chain code is R.

You can in turn derive keys from the obtained key and chain code to establish a key *hierarchy*. For example, from the key with index 0, you derive all your keys for Bitcoin, and from the key with index 60, you derive all your keys for Ethereum. Conventions associating a number to each blockchain network are standardized as part of the document "SLIP-0044: Registered Coin Types for BIP-0044."

If you first derive a child key with identifier 0 and next derive a key with identifier (say) 29 from this key and its chain code, then the key's *derivation path* is 0/29. You'll have done two calls of HMAC-SHA-512, yielding, respectively, L_1 and L_2 as their first 256 bits, and the final private key is $k + L_1 + L_2$. Therefore, you can see all keys derived from a given master key k as k plus the sum of values HMAC-SHA-512 returns. This derivation is called *hardened* because you need both the private key k and the chain code c of the parent key.

NOTE *For the* nonhardened *version, use the public key instead of the private key, which allows you to determine the public key of a child key from its parent public key. For more details, see the initial Bitcoin standard document "BIP32: Hierarchical Deterministic Wallets" and its generalized standard document "SLIP-0010: Universal Private Key Derivation from Master Private Key."*

Algebraic Hash Functions

Hash functions like SHA-3 and BLAKE3 operate on *bytes*, or on *words* of 4 or 8 bytes, where a byte is a chunk of 8 bits. The input data is a sequence of bytes, where each byte can take any of the 256 possible values from 0x00 to 0xff (255), and likewise the output data is a sequence of arbitrary bytes. This works well when using data efficiently converted into a sequence of bytes and when operations on bytes or words are efficient (such as XOR, word bit shift, and integer addition).

The Mathematical Computer

Say you have a computer that works with numbers only in a given range, such as numbers modulo 13. The XOR operation won't do well with such numbers, because not all 4-bit numbers are less than 13; for example, a XOR between 10 and 4 will give you 14, which is out of range. You might reduce it modulo 13 to obtain 1, but then 10 XOR 4 yields the same result as 10 XOR 11, a problem you don't have when working with bytes. This tends to significantly reduce the security of the hash function—through collisions, for example.

Even worse, your computer knows only operations modulo 13: addition, subtraction, multiplication, and division. It doesn't have a built-in bitwise XOR instruction, so it simulates it with its modulo 13 arithmetic instructions, which isn't straightforward or computationally efficient. I'll leave the details of this simulation as an exercise.

Your task is to create a hash function that works only (or mostly) with arithmetic operations for the given range of numbers and doesn't use bitwise operations including XOR, OR, AND, or bit shifts.

You need this kind of function to efficiently run certain advanced cryptographic protocols—namely, multiparty computation (MPC) and zero-knowledge proofs, which you'll see later in this chapter. Such protocols often operate in the realm of mathematical structures such as finite fields (for example, sets of integers modulo a prime number) and sometimes have

to "convert" a program into mathematical equations. In principle, you can convert any program into equations. But when the program isn't optimized for the underlying math structure, the equations get very large and slow to compute. *Algebraic hash functions* aim to address this issue by designing hash functions that are both secure and easy to implement with only arithmetic operations in a finite field.

Design Principles

Let's consider the design principles of *Poseidon*, a hash function designed in 2019 for zero-knowledge proof systems and rapidly adopted by many blockchain systems. Such proofs sometimes have to express a hash function as a circuit of arithmetic operations over a large finite field, such as those of integers modulo a 255-bit prime number. In that case, Poseidon proved orders of magnitude more efficient than general-purpose hash functions like SHA-256.

Poseidon uses the sponge hash function construction (see Chapter 7), so it needs to construct a *permutation*, an invertible transformation with input and output of the same size. It applies this permutation to a state that is a vector of finite field elements. In the relevant applications, such a finite field usually consists of numbers modulo a prime, which may be as small as 31 bits and as large as 381 bits, depending on the application.

Then, like most hash functions, the permutation iterates a series of rounds, so it needs to design a *round function*.

Finally, Poseidon breaks down its permutation into three layers, with three different purposes, reminiscent of AES's rounds:

- A *unicity* layer, *AddRoundConstants*, which Poseidon notes as *ARC*() in the documentation. This adds constant values to the state's elements, such that the constants are different for each round. Making each round unique prevents attacks including the slide attack. To avoid defining and storing many constants, Poseidon generates constants from a deterministic random bit generator initialized with an encoding of the Poseidon instance characteristics (number of rounds, finite field, S-box type, and so on).
- A *nonlinear* layer or *S-box layer*, *SubWords*, which Poseidon's documentation notes as *S*. This layer adds *confusion*, the property that the input and output values of the function are related by high-degree algebraic equations—and thus are as far as possible from linear and low-degree equations, which differential cryptanalysis can exploit. S-box transforms each element of its state independently of the others, typically mapping a field element x to x^3 or x^5. The exponent is kept relatively low to be efficiently computed.
- A *linear* layer, *MixLayer*, which Poseidon notes as *M*() in the documentation. MixLayer brings *diffusion*, or the propagation of differences in the initial state across all elements. For example, if the state consists of the four-element vector (x_1, x_2, x_3, x_4), then *M*() replaces each element with a linear combination of all elements. It might replace x_1 with

the result of $2x_1 + 10x_2 + x_3 + 3x_4$. Such a transformation corresponds to a multiplication of a vector by a matrix. Poseidon's matrices must satisfy certain security properties and should be designed for efficient implementation.

A *full round* of Poseidon applies the three layers in this order: *ARC*(), *S* to each element, and *M*(). A *partial round* applies *S* to only one element and may use a different matrix in *M*(). A Poseidon *instance* then iterates full rounds, partial rounds, and full rounds—the number of which depends on the instance, number of elements, finite field, and target security level.

NOTE *For more details on Poseidon, see* https://www.poseidon-hash.info *as well as the initial Poseidon paper (*https://eprint.iacr.org/2019/458.pdf*) and the improved design Poseidon2 paper (*https://eprint.iacr.org/2023/323.pdf*).*

Poseidon is one of many algebraic hash functions created to address practical use cases. Other designs include MiMC, Monolith, Rescue-Prime, and Tip5.

How Things Can Go Wrong

Let's look at some security failures that involve hash functions and their applications in the blockchain world.

Broken Custom Hash

The 2017 blockchain project Iota was quite weird. It claimed to use an architecture different from the sequential chain of blocks of most blockchains, closer to direct acyclic graph (DAG), which was much less secure. It also encoded data not in bits but in *trits*, units taking three values instead of two, supposedly to make computations more efficient on processors that don't exist.

Iota didn't use an elliptic curve-based signature scheme like ECDSA or Ed25519 but rather a hash-based signature scheme based on the established Winternitz construction, thus offering post-quantum security—except Iota also designed its own custom hash function, Curl.

Iota became one of the top 10 most popular cryptocurrencies and made grandiose claims about its potential usefulness and its security. But its custom hash function, which it claimed to develop with the aid of artificial intelligence, turned out to be very weak against collision attacks. Using off-the-shelf hardware, researchers found collisions within minutes, which could be exploited only in contrived attack scenarios. Iota quickly patched its hash function.

After this fiasco, renowned cryptography and security expert Bruce Schneier commented, "In 2017, leaving your crypto algorithm vulnerable to differential cryptanalysis is a rookie mistake. It says that no one of any caliber analyzed their system, and that the odds that their fix makes the system secure is low."

Wallet with Low Entropy

The hierarchical key derivation model from earlier in this section is secure on paper but in practice only if it's implemented correctly. You can usually verify this using test vectors within the BIP32 and SLIP-0010 standard documents. If you obtain the same input/output values as those documented, your implementation is likely correct, though not necessarily secure.

In 2022, the popular cryptocurrency mobile wallet application Trust Wallet announced the release of a browser extension version that used WebAssembly (Wasm) technology to run efficiently on different browsers. However, Wasm couldn't use the same PRNG as the mobile versions; it had to define a different one.

A poor PRNG can be cryptography's Achilles' heel (see Chapter 2). In Trust Wallet, the developers used the Mersenne Twister PRNG (mt19937), which is not a cryptographic PRNG. Its entropy is at most 32 bits, and it produces its output bits using simple linear combinations of the internal state values.

As Trust Wallet's PRNG had 32 bits of entropy, it could generate only 2^{32} different seeds for users' wallets. An attacker could then compute all the 2^{32} possible seeds and, for each seed, compute the private keys and addresses derived using hierarchical key derivation. They could then scan blockchains to find addresses generated by Trust Wallet and steal their tokens. The Ledger company researchers who found the flaw commented, "Running such an attack takes much more than a couple of hours, but is doable with a few GPUs in less than a day."

Collisions from Domain Separation Failures

Let's discuss how to find collisions for a hash function that is collision resistant. As cryptographer Moti Yung said, "If it sounds impossible, then it's cryptographically interesting."

Imagine the following simple case: an application receives messages A and B from two parties, Alice and Bob, and then hashes these two messages together to create a hash value unique for these messages. It could then compute $\mathbf{H}(A \,||\, B)$ by hashing the string consisting of A followed by B. Even if your hash function is collision resistant, you'll get the same hash value for A = COL and B = LISION as for A = CO and B = LLISION. You end up with a hash collision with respect to the application's input values, although it's not a collision for the hash function because the hash function **H** processes the same string in the two cases, COLLISION.

To avoid the problem, encode the application's input values into a string that's unique per input, with no ambiguous encoding: for each string, you should be able to uniquely identify the original set of input values. In our example, adding a dollar sign ($) character as a separator between the two inputs seems to avoid the collision between COL || LISION and CO || LLISION: you would get the strings COL$LISION and CO$LLISION, leading to different hash values.

But a separator symbol isn't enough to eliminate ambiguities when the character is authorized in the application's input values. For example, take the string COL$$LISION as the concatenation of two input strings separated with $. You could have obtained that string from two pairs of inputs: COL and $LISION, COL$ and LISION. Note that the problem doesn't occur when the input values are a fixed, constant size—for example, a first string of two characters and a second of three characters. Even in that case, it's still safer to use some separator or encoding that prevents collisions, as a future patch may introduce variable-length input.

Researchers have found bugs of this type in threshold signing protocols, which you'll see later in this chapter, as well as in e-voting protocols, where the resulting hash collisions could be exploited to violate the security properties of these protocols.

Multisignature Protocols

In a cryptographic *multisignature protocol*, participants jointly produce a signature of a message that's functionally equivalent to having all parties separately sign the message; an obtained signature means all parties agreed to sign the message. The advantage is that instead of having as many signatures as signers, there's only one. Anyone who has the public keys of all the signers can then verify the signature.

Blockchain platforms employ multisignatures when multiple parties manage an account to ensure that all parties endorse issued transactions. A single party can also use them if they have multiple keys on multiple devices to prevent a single compromised key from allowing the attack to issue transactions. Such multisignature protocols are one of the many types of *collective signature* protocols, where parties run a protocol to produce a signature.

Before diving into the technical details, let's clarify how these multisignatures differ from related protocols.

Multiple Multiparty Signatures

Despite their similar name, the multisignature protocols we'll discuss are different from on-chain multisignature scripts or multisignature smart contracts, as used in Bitcoin and Ethereum, respectively. The latter aren't cryptographic protocols because participants independently submit their individual signature to the blockchain network. The network in turn verifies a rule such as "if a transaction has signatures from pub_1 and pub_2, then accept it" or "if a transaction is signed by any two parties among pub_1, pub_2, and pub_3, then accept it"—rejecting the transaction otherwise. Here we implicitly view public keys *pub* as the identifiers of the parties, as is common in blockchain protocols.

Compared to on-chain multisigs, multisignature protocols produce a single signature; there's then only one signature to verify rather than multiple ones. The multisignature scripts and smart contracts instead process several signatures and consist of a verification rule rather than a protocol

on the signers' end. In both cases, verification requires the public keys of all signers.

Multisignature protocols also differ from two types of other collective signature protocols, which you'll see in the following sections. In both protocols, the outcome is a single signature, but the difference is how and from what it's created:

Aggregate signature protocols

- Like in multisignatures, each participant has their own key pair (public and private keys).
- Unlike in multisignatures, participants may sign different messages, rather than the same one. A participant can also sign multiple messages.
- Like in multisignatures, the verification of a signature requires multiple public keys (or a single aggregate version thereof, for supporting protocols).

Threshold signature protocols

- Unlike in multisignatures, participants don't use their own keys. Instead, they have shares (also called *shards*) of a single private key, such that no single participant ever knows the full key, even during the protocol's execution.
- Like in multisignatures, participants sign a single message.
- Unlike in multisignatures, verification requires a single public key.

Now that we've defined what multisignatures are, let's see how they work in their most illustrious use case: Schnorr signatures.

Schnorr Signature Protocols

Mathematician Claus-Peter Schnorr created the eponymous signature scheme in 1989 and filed a patent for it, which prevented its wide adoption until 2008, when the EdDSA scheme (see Chapter 12) optimized it to work with modern elliptic curves. Schnorr's scheme is simpler than the ECDSA standard, making it easier to turn into a multisignature scheme. Bitcoin supports Schnorr signatures, which were introduced in 2022 as better support for multisignature protocols.

NOTE *We'll use additive notation (as with EdDSA and when working with elliptic curve), rather than the original multiplicative notation (as used when working with integers in a multiplicative group). Therefore, the public key* A *associated with a private key* a *is the elliptic curve point* $A = aG$*, where* G *is a predefined base point, and group elements are points combined by addition. This is opposed to the multiplicative notation where we'd have* $A = g^a$ *for some group generator* g*, where group elements are numbers multiplied together.*

Single-Signer Schnorr Signatures

Before seeing how multiparty signing works, let's see how single-party Schnorr signing works. Suppose Alice has a private key a, with associated public key $A = aG$. Here a is a number, or scalar, in a given range of numbers (specifically, the finite field over which the elliptic curve is defined, typically positive integers modulo some large prime numbers, of at least approximately 256 bits), and G is a fixed base point of the curve.

To sign a message M, Alice proceeds as follows:

1. Pick a secret random number, r, and compute the point $R = rG$. The value r is a *nonce*, a one-time private key with R its public key.
2. Compute $h = \mathbf{H}(R \,||\, A \,||\, M)$, the value that you'll "connect" to the private key a and the one-time private key r to sign M. We not only hash the message but also bind h to the signer and the nonce, respectively, via the nonsecret values A and R. Without these, different attacks would be possible.
3. Compute $s = r + ha$ and return the pair (R, s) as a signature. You can see s as the multiplication between the secret key and the data to sign, where the secret r masks the result; without this, it's trivial to recover the private key from a signature.

Verify a signature by checking that sG equals $R + \mathbf{H}(R \,||\, A \,||\, M)A$. This works because from $s = r + ha$, substituting $r + ha$ for s in sG, you obtain

$$sG = (r + ha)\,G = rG + haG = R + hA$$

with $h = \mathbf{H}(R \,||\, A \,||\, M)$, which the verifier must compute from the message M, the public key A, and the R part of the signature.

Schnorr Multisignatures

In multisignatures we don't have one signer but multiple ones. For the sake of simplicity, we'll describe the case of two cosigners: meet Bob, who'll sign messages jointly with Alice. Bob's private key is b and his public key is $B = bG$. To jointly create a multisignature, Alice and Bob could proceed as follows to sign a message M:

1. Alice picks a nonce r_A, computes $R_A = r_A G$, and sends R_A to Bob.
2. Bob picks a nonce r_B, computes $R_B = r_B G$, and sends R_B to Alice.
3. They compute $R = R_A + R_B$ and set $h = \mathbf{H}(R \,||\, A \,||\, B \,||\, M)$, the value that Alice and Bob will use to generate their part of the signature. A specific value of h is bound to the parties via their public keys (A and B) and bound to the current signing session via the nonce R only for the specific signature execution as defined by the nonce R.
4. Alice computes $s_A = r_A + ha$ and sends it to Bob.
5. Bob computes $s_B = r_B + hb$ and sends it to Alice.

6. They both compute $R = R_A + R_B$ and $s = s_A + s_B$ and return (R, s) as the signature.

To verify a signature, check that sG equals $R + h(A + B)$. Substituting s in sG by the previously computed value results in the following:

$$sG = (s_A + s_B)\,G = (r_A + ha + r_B + hb)\,G$$
$$= (r_A + r_B)\,G + h(a + b)\,G = R + h(A + B)$$

Note that the verifier needs to know both A and B, and not only their sum $A + B$, because they need both values to compute $\mathbf{H}(R \,||\, A \,||\, B \,||\, M)$. However, if h is instead defined as $\mathbf{H}(R \,||\, A + B \,||\, M)$, a verifier can use a single public key $A + B$ without knowing that the signature is issued by two parties. We call the "merging" of multiple public keys into one *key aggregation*. This is particularly useful to reduce the size of the data to hash when there are many signers.

NOTE *I've described basic Schnorr multisignatures in the case of two parties, but the protocol scales to an arbitrary number of parties with public keys* $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n$. *In the definition, replace* $\mathrm{A} + \mathrm{B}$ *with* $\mathrm{P}_1 + \mathrm{P}_2 + \ldots + \mathrm{P}_n$, *replace* $\mathrm{A} \,||\, \mathrm{B}$ *with* $\mathrm{P}_1 \,||\, \mathrm{P}_2 \,||\, \ldots \,||\, \mathrm{P}_n$, *and replace "sends it to Bob/Alice" with "sends it to everyone." You can apply a similar protocol to EdDSA and Ed25519, variants of Schnorr's scheme.*

How Things Can Go Wrong

The Schnorr multisignature protocol is relatively simple but may fail in the following attack scenarios.

Key Cancellation Attack

In this attack, Bob convinces signature verifiers that he jointly signed a message with Alice, whereas Alice hasn't seen the message and didn't interact with Bob. An attacker could exploit this in a scenario where you expect Alice and Bob to jointly sign messages—for example, in transactions that require both parties' approval. In the normal case, the verifiers would know Alice's public key A and Bob's public key B, and Bob and Alice would know each other's keys.

Now suppose the following happens: Alice sends her public key A to everyone including Bob, but instead of sharing his public key B, Bob shares $C = B - A = (b - a)G$ with the verifiers, and B with Alice. That Bob doesn't know the private key corresponding to C won't matter for the attack.

Bob has to sign a message using his private key b, as in the single-signer case, but with $h = \mathbf{H}(R \,||\, A \,||\, C \,||\, M)$, as if he were signing with Alice. He returns (R, s) as a signature, with $R = rG$ for an r of his choice and $s = r + hb$.

Expecting signatures from Alice and Bob, verifiers check that sG equals $R + h(A + C)$, which is correct because $A + C = A + (B - A) = B$. Bob can therefore forge multisignature without even interacting with Alice and without knowing her private key.

In practice, you can avoid this attack by requesting signers to prove the knowledge of the private key—for example, by signing a message. Since Bob doesn't know the private key corresponding to C, he can't provide such evidence. As you'll see with the MuSig protocol, you can also avoid the attack at the protocol level.

The attack scales to an arbitrary number of parties, a context that's often called a *rogue key attack*. Having received the public keys of all other parties, an attacker Bob would just define his public key as $B - X$, with X the sum of all other parties' public keys.

Repeated Nonces

Like in ECDSA, repeated nonces are lethal to the Schnorr multisignature protocol. Imagine that Alice's pseudorandom generator fails and she generates the same secret nonce r_A twice in two runs of the protocol: she sends an initial $s_A = r_A + ha$ to Bob, and a second $s_A' = r_A + h'a$ where the first h and the second h' also depend on Bob's randomness. You thus have $r_A = ha - s_A$ and $r_A = h'a - s_A'$, which implies $ha - s_A = h'a - s_A'$ or, equivalently,

$$a(h - h') = s_A - s_A'$$

from which you can compute Alice's private key $a = (s_A - s_A') \,/\, (h - h')$.

One approach to eliminate security risks caused by a randomness failure is to get rid of randomness. For example, computing the nonce by hashing the message and the private key as in Ed25519 works when there's a single signer. However, setting r to $\mathbf{H}(a \,\|\, M)$ won't work in the case of multisignatures: if Alice and Bob sign the same message twice, then Alice computes a first $s_A = r_A + ha$ and a second $s_A' = r_A + h'a$, where in both cases $r_A = \mathbf{H}(a \,\|\, M)$, where h' will be distinct from h if a malicious Bob sends a different value than $\mathbf{H}(b \,\|\, M)$. In this case, Bob—and any eavesdropper of the communications—can again compute a as $(s_A - s_A') \,/\, (h - h')$.

Parallel Execution Insecurity

The Schnorr multisignature protocol is insecure when the attacker can initiate multiple simultaneous signature protocols. The attack is too complicated to describe here but is documented in the research articles available at *https://eprint.iacr.org/2018/417* and *https://eprint.iacr.org/2020/945*.

Safer Schnorr Multisignatures

To avoid the key cancellation attack and repeated nonce issues, researchers developed more advanced multisignature protocols and in particular the MuSig protocols: MuSig, MuSig2, and MuSig-DN, where *MuSig* stands for *multisignature* and *DN* for *deterministic nonce*. MuSig protocols also support key aggregation to allow a verifier to check a signature using only one public key that's derived from the signers' keys, such that the aggregated key doesn't reveal the number of signers or their public keys.

Let's see MuSig's main trick in action. If we're in the simplest case of two signers, Alice and Bob, using the same notations as in the previous sections, instead of computing $s_A = r_A + ha$ as her share of the signature, Alice computes $s_A = r_A + \mu_A ha$, thus multiplying the ha part with the μ_A value. She computes μ_A (where μ is the Greek letter mu) by hashing the list of participants' public keys followed by Alice's key, $\mathbf{H}(A \parallel B \parallel A)$. Likewise, Bob computes $s_B = r_B + \mu_B hb$ with $\mu_B = \mathbf{H}(A \parallel B \parallel B)$, hashing the list of public keys followed by his key.

Alice then computes the *aggregate public key* as $X = \mu_A A + \mu_B B$, the sum of the public keys multiplied by their respective μ values. The hash of the message is then $h = \mathbf{H}(R \parallel X \parallel M)$ instead of $\mathbf{H}(R \parallel (A + B) \parallel M)$ in the vulnerable version multisignature scheme supporting key aggregation.

This trick works because a malicious Bob can no longer forge another public key that would "cancel" Alice's A, as he could by setting $C = B - A$ in the key cancellation attack. In the equation $X = \mu_A A + \mu_B B$, Bob has to find a new value of B that yields the "right" μ coefficients to remove A from the equation. But that's now impossible because the equation is nonlinear with respect to A and B (linearity is often synonymous with insecurity—see Chapter 2).

When there are more than two signers, you apply the trick in a similar way, computing the μ coefficients by hashing the list of keys followed by the signer's key and then aggregating the public keys into a single X by computing the sum of the public keys multiplied by their respective μ.

NOTE *For more details on the MuSig protocols and how the MuSig-DN version securely derives nonces from the message, see* https://bitcoinops.org/en/topics/musig/.

Aggregate Signature Protocols

Aggregate signatures have multiple signers, and each signer signs a message (which can be distinct for all signers); then these signatures are merged into a single signature. From only this signature and from the signers' public keys and the messages they signed, verification checks that all signers signed their respective messages. Because a single signature must be stored, instead of as many signatures as signers, verification time is proportional to the number of messages. When all signers sign the same message, verification can be as fast as the verification of a single signer's signature, regardless of the number of signers.

Aggregate signatures are notably used in Ethereum, specifically in its consensus layer. In this use case, validator nodes endorse proposals to change the state of the system (as blocks) and leverage aggregate signatures to minimize the signature storage space and verification time. They use the Boneh–Lynn–Shacham (BLS) signature scheme, which you'll learn about in this section, starting with the magic behind BLS signatures: cryptographic pairings.

Pairings

In elliptic curve cryptography, a *pairing* is an operation that transforms two points from two elliptic curve groups (not necessarily the same) into a finite field element. The standard notation for a pairing between two elliptic curve points P and Q is $e(P, Q)$. Pairings used in cryptography have the property called *bilinearity* and are thus called *bilinear pairings*, which means they satisfy the following for any points P, Q, and R:

$$e(P + R, Q) = e(P, Q) \times e(R, Q)$$
$$e(P, Q + R) = e(P, Q) \times e(P, R)$$

Here, adding a point R to an operand is equivalent to multiplying the result by the pairing between R and the other operand. Consequently, if you add a point to itself n times—that is, if you multiply it by a number n—then you have

$$e(nP, Q) = e(P, Q) \times e(P, Q) \times \ldots \times e(P, Q) = e(P, Q)^n$$

or $e(P, Q)$ multiplied by itself n times, which is also equal to $e(P, nQ)$.

If you have different points $P_1, P_2, \ldots, P_n$, you can turn addition of the input values into multiplication of the output values:

$$e(P_1 + P_2 + \ldots + P_n, Q) = e(P_1, Q) \times e(P_2, Q) \times \ldots \times e(P_n, Q)$$

The internals of how pairings work is beyond this book's scope. For more details, see the article "Cryptographic Pairings" by Kristin Lauter and Michael Naehrig (*https://eprint.iacr.org/2017/1108*) and the book *Guide to Pairing-Based Cryptography* by Nadia El Mrabet and Marc Joye (Chapman and Hall/CRC, 2016).

BLS Signatures

BLS signatures were presented in 2006 in the article "Short Signatures from the Weil Pairing" by Dan Boneh, Ben Lynn, and Hovav Shacham. It stated that the scheme was designed "for systems where signatures are typed in by a human or signatures are sent over a low-bandwidth channel." BLS signatures turned out to be used in highly automated systems over high-bandwidth channels, which benefited from a property described in a subsequent paper: aggregation of signatures and of public keys.

NOTE *Don't confuse BLS signatures with BLS (Barreto–Lynn–Scott) curves, which have an author (Lynn) in common. BLS curves are elliptic curves designed to be pairing friendly, allowing secure and efficient pairing operations. In fact, BLS signatures often work with points over a BLS curve. For example, the Ethereum BLS signature relies on the curve BLS12-381.*

Single-Signer Signature and Verification

In BLS signatures, Alice's private key is a scalar number a, and her public key is $A = aG$, for a predefined base point G. To sign a message M, she first computes $H = \mathbf{H}(M)$, where the hash function returns a curve point rather than a bit string or a scalar—the H notation follows the general convention that you denote points as capital letters. The signature is then $S = aH$. This looks much simpler than a Schnorr's or an ECDSA signature: just hash the message and multiply the result with the private key.

To verify a BLS signature, compute two pairing operations:

- $e(A, H)$ between the public key and the hashed message (note that $A = aG$)
- $e(G, S)$ between the base point and the signature (note that $S = aH$)

These values should be equal because of the pairing's bilinearity property:

$$e(A, H) = e(aG, H) = e(G, H)^a = e(G, aH) = e(G, S)$$

If the equality holds, the signature is accepted; otherwise, it's rejected.

Ignoring the complexity of the pairing operation, such pairing-based signatures are the simplest signature scheme.

Aggregate Signatures from Multiple Signers

Let's further exploit the magic of BLS signatures and bilinear pairings, considering a scenario wherein n signers with private keys $k_1, k_2, \ldots, k_n$ and public keys $P_1, P_2, \ldots, P_n$ sign n messages $M_1, M_2, \ldots, M_n$ and produce the signatures $S_1, S_2, \ldots, S_n$. Note that $H_i = \mathbf{H}(M_i)$ is the hash of the ith message.

You can aggregate signatures $S_i = k_i H_i$ into one by adding them to obtain $S = S_1 + S_2 + \ldots + S_n$. Observe that computing $e(G, S)$, as in the verification of a single signature, results in the following, owing to the bilinearity property:

$$e(G, S) = e(G, S_1 + S_2 + \ldots + S_n) = e(G, S_1) \times e(G, S_2) \times \ldots \times e(G, S_n)$$

Remember that pairings satisfy $e(nP, Q) = e(P, nQ)$. You can thus replace each $e(G, S_i)$ term with $e(P_i, H_i)$ by "moving" the multiplicative factor k_i to the left operand of the pairing: the first operand will thus be $k_i G = P_i$ instead of G, and the second will be H_i instead of $S_i = k_i H_i$.

After aggregating multiple signatures of multiple signers over multiple messages into a single signature, you can verify the aggregate signature from the signers' public keys and messages signed. Verification just checks the equality between $e(G, S)$ and the product of all pairings $e(P_i, H_i)$. There are $1 + n$ pairings to compute, rather than the $2n$ pairings if you hadn't aggregated the signatures—in this case, the signatures occupy n times as much memory as the aggregated one.

Let's consider a scenario where you'll aggregate both signatures and public keys.

Aggregate Public Keys

Imagine that all signers sign the same message M and that you aggregate all public keys into a single one: $P = P_1 + P_2 + \ldots + P_n$. Note that $k_1, k_2, \ldots, k_n$ still represents their respective private keys and $H = \mathbf{H}(M)$ the hash of the message. Given valid signatures S_i, you obtain the following equality:

$$\begin{aligned} e(P, H) &= e(P_1 + P_2 + \ldots + P_n, H) \\ &= e\big((k_1 + k_2 + \ldots + k_n)\, G, H\big) \\ &= e\big(G, (k_1 + k_2 + \ldots + k_n)\, H\big) \\ &= e(G, S_1 + S_2 + \ldots + S_n) \\ &= e(G, S) \end{aligned}$$

You therefore verify the signatures of n parties over the same message using only two pairing operations, $e(P, H)$ and $e(G, S)$. This is extremely efficient, as it makes signature verification essentially independent of the number of parties, as you can add points efficiently, whereas pairings are costly to compute. In addition to the computing efficiency, aggregating keys and signatures also saves memory.

How Things Can Go Wrong

Like many elliptic curve cryptography schemes, BLS signatures should avoid invalid keys and weak parameters to be secure. And like the previous protocols, BLS signatures could be vulnerable to key cancellation attack. Let's explore the details.

Invalid Keys

BLS signatures are specified in an Internet-Draft, a working document of the IETF available at *https://github.com/cfrg/draft-irtf-cfrg-bls-signature*. This specifies core operations including key generation (the algorithm `KeyGen`), signature (`CoreSign`), verification (`CoreVerify`), and key validation (`KeyValidate`). Given a public key, the latter ensures the validity of a public key in that it "represents a valid, non-identity point that is in the correct subgroup."

Key validation prevents the use of weak private/public keys pairs, for which signatures are easier to forge. For example, take the trivial case of a zero secret key $a = 0$. It follows that the signature of any message M is $0 \times \mathbf{H}(M) = 0$. It's therefore trivial to forge a signature for any message. In that case, the public key is then $0 \times G = O$, which is the point at infinity. If key validation rejects public keys equal to O, it ensures that the secret key is not zero.

A less trivial case is when the public key, as an elliptic curve point, has a value that makes it easier to forge signatures—that is, to create a valid signature without knowing the private key. Not all points on a given elliptic curve are equally secure—in particular, points that belong to a small

subgroup rather than to the main subgroup of points. If a public key point belongs to one such small subgroup, there will be way fewer possible valid signatures, making it easier to forge a signature. Likewise, if the public key offered to the verification function doesn't belong to the elliptic curve, then valid signatures are easy to forge.

It's therefore crucial to check that public keys are valid using the `KeyValidate` algorithm from the aforementioned specification, as copied in Listing 15-1.

```
Inputs:
- PK, a public key in the format output by SkToPk.

Outputs:
- result, either VALID or INVALID

Procedure:
1. xP = pubkey_to_point(PK)
2. If xP is INVALID, return INVALID
3. If xP is the identity element, return INVALID
4. If pubkey_subgroup_check(xP) is INVALID, return INVALID
5. return VALID
```

Listing 15-1: The `KeyValidate` algorithm ensures that a BLS public key is valid.

If you implement BLS signatures, make sure your code does all the checks described in the BLS specification when validating public keys and signatures.

Key Cancellation Attack

In their basic form, BLS aggregate signatures with aggregate public keys are subject to the same type of key cancellation attack as Schnorr signatures: if an attacker knows the public keys $P_1, P_2, \ldots, P_{n-1}$ of the first $n-1$ signers, they can claim that their public key is

$$P_n = X - (P_1 + P_2 + \ldots + P_{n-1})$$

where X is the public key for which they know the private key x, such that $X = xG$. When the attacker presents a signature created from x, unsuspecting users verify a message's signature using the public key $P_1 + P_2 + \ldots + P_n$, which is equal to X. The attacker can single-handedly sign a message on behalf of the set of supposed signers.

To prevent this attack, users may prove the knowledge of their public key's private key by signing a message. As the attacker doesn't know the private key of P_n, they would fail this test.

Another mitigation is to modify the aggregate signature scheme so that the returned signature isn't just the sum of signature $S = S_1 + S_2 + \ldots + S_n$ but instead the sum with coefficients derived from public keys, as in

$$S = t_1 S_1 + t_2 S_2 + \ldots + t_n S_n$$

where $t_i = \mathbf{H}(P_i \| P_1 \| P_2 \| \ldots \| P_n)$ for $i = 1, 2, \ldots, n$. The aggregate public key used to verify a signature is then $P = t_1P_1 + t_2P_2 + \ldots + t_nP_n$. You can see that this trick works by checking that $e(P, \mathbf{H}(M))$ still equals $e(G, S)$.

Threshold Signature Protocols

Threshold signatures differ from multisignatures and aggregate signatures—which require all participants in the signing protocol to have their own public and private key—in that there's a single private key k and a single public key P, and there are n participants that each has a distinct *share* of the key k_i where a parameter t (the *threshold*) is defined such that $t < n$ and:

- $t + 1$ signers can jointly issue a valid signature of some message that is verified using the public key P, in such a way that no signer learns the private key k. This is done by running a protocol that uses the shares k_i of each signer and the message to be signed, without ever exposing the private key to any party.
- A set of t signers or fewer can't create a signature and therefore can't determine the private key k either.

The issued signature looks like a normal, single-signer signature and verifies as such.

Threshold signatures are a specific type of *multiparty computation (MPC)*, a class of protocols wherein n participants compute the output of some function $f(x_1, x_2, \ldots, x_n)$ such that each participant knows their respective input x_i and learns the output of the function but doesn't learn the other participants' x_i inputs. In the case of threshold signatures, the x_is are shares of the private key, and the output is a signature. I'll elaborate on what a share is in the "Secret-Sharing Techniques" section on page 319.

Threshold signatures have the benefit of "hiding" the number of cosigners and their identities, as verifiers see only a signature from a single private key. While some multisignature and aggregate signature schemes share this property, threshold signatures better suit the use case of crypto asset custody because there's only one private key and one public key; you can directly apply threshold signatures to distribute control of any address.

Use Cases

Threshold signatures are used in cryptocurrency and digital asset management to distribute control of an address across multiple systems or parties. They can serve to share control of an account between a service provider and a user's device: each party has one share of the key and must run a protocol together to sign a transaction, thus spending funds. This setup ensures that an attacker can't autonomously authorize transactions, even if they breach the user's device and obtain the key shares. Similarly, the provider can't initiate transactions without the user's consent. However, implementing this model reliably poses challenges, particularly key management aspects (key generation, key rotation, backups, and so on).

An organization can also use threshold signatures to distribute custody of funds across multiple systems, such as different device types, data centers, operating systems, and software components. This approach is particularly suitable for cold wallets and accounts containing significant assets. Relying solely on threshold signatures is insufficient, as comprehensive security measures and controls are crucial. For example, one must properly separate accesses to IT components to ensure that distinct individuals or IT service providers have access to different systems (thus different shares). Moreover, the initiation and approval of transactions must be subject to strict controls and an audit trail.

Security Model

As with all cryptographic protocols, we need to define what it means for a threshold signing scheme to be secure. Such a security model includes the security goal (what should be hard for an attacker to do) and the attacker model (the assumptions about the attacker's capabilities). Let's delve into these two characteristics.

Security Goals

The main security goal of threshold signature is the same as that of single-party signatures: an attacker must not be able to forge valid signatures, which implies they must not be able to determine the private key. This also implies that the protocol ensures *input privacy*: the key shares held by the parties shouldn't leak to any other party. Finally, the protocol must ensure *correctness*: the signature the protocol computes must be valid and accessible to all participants in the protocol execution.

Attacker Models

There isn't one security model for threshold signatures; instead, an attacker is characterized according to several dimensions. The common threat in all attack scenarios is the assumption that an attacker can actively attack network communications—capturing, modifying, and injecting messages. Using a secure channel to establish authenticated and encrypted communications between the parties thwarts such attacks. Protocol designers also assume reliable communications in that all transmitted messages should be received in the same order they were sent.

The attacker model of threshold signatures considers the *corruption* of participants—namely, that an attacker compromises their system, learns their secrets, and essentially gets them to do what they want. The model assumes that an attacker can't corrupt more than t parties; otherwise, they'd be able to forge signatures, by definition of the threshold signing functionality.

Let's examine the parameters to consider when an attacker can corrupt participants.

First, we characterize an attacker by the *number of parties* they could corrupt. Two categories of threshold signatures exist, each defined by the

maximum number of malicious participants that can be compromised without compromising the security of the protocol:

Honest majority Here, the attacker is limited to corrupting less than half of the key-sharing parties. Therefore, the threshold t must satisfy $t < n/2$. By definition, a threshold signing protocol with parameters (t, n) must be safe even with compromised t participants. Protocols designed under the honest majority assumption are typically more efficient, but they can't accommodate arbitrary values of t. For instance, a protocol with parameters (4, 5) is infeasible in this model, as it requires tolerating up to four corrupted parties.

Dishonest majority This model allows the protocol to support any value of t from 1 to $n - 1$. It enables the creation of protocols where all but one of the n parties could be corrupted, yet the malicious parties would still be incapable of forging signatures or recovering the private key. This model offers more flexibility in terms of the threshold value but often requires more complex and robust security mechanisms.

We also characterize an attacker by what they can do once they corrupt a party and learn their secret values (including the key share). There are two attacker models defining this:

Passive, or honest-but-curious They learn information from corrupted parties but can't force them to deviate from the protocol. This models "read-only" compromises, where an attacker gets a snapshot of a system's memory, both storage and volatile memory (RAM, processor registers).

Active, or malicious Parties may arbitrarily deviate from the prescribed protocol. This models systems fully compromised by an attacker or controlled by a malicious insider (such as an operator, administrator, or cloud provider).

A protocol secure against active attackers is therefore always secure against passive attackers, but not the other way around.

There are two ways for an attacker to choose which parties to corrupt, which are defined by the following models of corruption type:

Static corruption The attacker must choose which participants to corrupt before starting the protocol.

Adaptive corruption The attacker may wait until after the protocol begins to choose which participants to corrupt—and then learn the history of their operations throughout the protocol.

A protocol secure against static corruption is always secure against adaptive corruption, but not the other way around. However, there are techniques to convert a protocol from secure against static corruption to secure against adaptive corruption.

Secret-Sharing Techniques

A key ingredient of threshold signatures is *secret-sharing* protocols, or techniques that split a secret into multiple pieces (*shares*) distributed to different parties in a way that the parties can then jointly recompute the initial secret. Secret-sharing protocols can notably be used when creating backups of private keys such that different parties store different shares in different locations.

Additive Sharing

The simplest way to share a secret that you see as a number is through *additive sharing*: given a number s, you share it as n values $s_1, s_2, \ldots, s_n$ such that $s_1 + s_2 + \ldots + s_n = s$. For example, if you work with numbers modulo 100, you could create a random additive sharing of the number $s = 47$ into four shares as follows: pick three random numbers between 0 and 99, say $s_1 = 12$, $s_2 = 94$, and $s_3 = 80$, and set $s_4 = s - s_1 - s_2 - s_3 = 61$. (Note that subtraction is computed modulo 100, such that $-1 = 99$, $-2 = 98$, and so on.)

This approach is very simple but requires using all the shares to recover the original secret.

Threshold Sharing

Threshold sharing is closer to what threshold signatures do: given parameters n and $t < n$ and the secret s, it creates shares such that you can recover the secret using any set of t shares out of the n created.

The best-known threshold sharing method is *Shamir's secret sharing*, which leverages the following property of polynomials: given a polynomial of degree t of the form

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_tx^t$$

you need the result of only $t + 1$ evaluations of $f(x)$ on $t + 1$ distinct values of x to determine all a_i coefficients, which are fixed values.

To create a threshold secret sharing from this property, set $a_0 = s$, the secret. Then pick random values of a_1 to a_t and compute $f(x)$ for n distinct values of x, which will be the n shares of the secret.

Recomputing the coefficients from the $f(x)$ values is a technique called *Lagrange interpolation*, from the 18th-century Italian mathematician who developed a general method to determine the equation of a curve given a list of points on that curve. Indeed, you can view the problem geometrically. If the equation has degree 1 (and is of the form $a_0 + a_1x$), then it's the equation of a line, and knowing two points on a line is sufficient to uniquely identify the line. Likewise, if the equation has degree 2 (of the form $a_0 + a_1x + a_2x^2$), then the curve is a parabola, whose equation can be determined with three points.

We sometimes find implementations of Shamir's secret sharing defining a function **Lagrange**() that computes the interpolation and returns the a_0 coefficient, which is the shared secret. If you have values of $f(x)$ to combine

to recover the secret, you can define the operation **Lagrange**(s_1, s_2, . . . , s_t) returning the shared secret s. This works for any combination of t distinct shares, not necessarily the first t ones. The details of the **Lagrange**() operation are a bit too technical for this book, but you can implement it as a series of basic additions, multiplications, and inverses.

The Trivial Case

One of the simplest types of threshold signature uses the BLS signature, as viewed earlier in the context of aggregate signatures. Recall that given a private key k, BLS signatures compute a signature by multiplying k with the curve point $H = \mathbf{H}(M)$. You can use additive sharing to create a threshold scheme with parameters $(n - 1, n)$. For example, if $n = 3$, split the key into three shares such that $k_1 + k_2 + k_3 = k$. Each party then computes their share of the signature by multiplying their share k_i with H. Adding the three shares then yields the following:

$$k_1H + k_2H + k_3H = (k_1 + k_2 + k_3)H = kH$$

Combining the three shares via addition gives a valid signature with k, even if none of the parties knows k. The parties could also recover k if they add their respective shares, but adding the shares of two parties reveals no information on k if the shares were randomly generated.

To create a threshold signature scheme with arbitrary t and n, use the Shamir secret-sharing technique and leverage the linearity of the **Lagrange**() operation: you generate the key shares k_i as described in the "Threshold Sharing" section and compute the signature like so:

$$\mathbf{Lagrange}(k_1H, k_2H, k_3H) = \mathbf{Lagrange}(k_1, k_2, k_3)H = kH$$

Again, you obtain a valid signature with the key k.

The Simple Case

You'll now compute Schnorr signatures in the threshold setting, which is a bit more technical than with BLS signatures. Recall the Schnorr scheme computes a signature as $s = r + ha$, where $h = \mathbf{H}(R \,\|\, A \,\|\, M)$, r is the per-signature random nonce, and a is the signer's private key. The signature also includes $R = rG$, the public nonce value. This is relatively easy to turn into a threshold signature scheme, owing to its linearity with respect to secret values: observe that the secret r is added to the secret a multiplied by the hash h, which is not secret.

Imagine the simplest case of two signers, with an additive sharing of the private signing key $k = a + b$, with a and b the respective key shares of the two signers. To sign, the two parties could generate secret nonces r_A and r_B, with respective public values $R_A = r_AG$ and $R_B = r_BG$. The parties can then exchange these values and compute the public nonce $R = R_A + R_B$, which will be part of the signature. R is then the public value you derive from the private value $r = r_A + r_B$, since you have:

$$R_A + R_B = r_AR + r_BR = (r_A + r_B)R$$

Next, the parties compute their shares of the signature: $s_A = r_A + ha$ and $s_B = r_B + hb$, which are added together to yield:

$$(r_A + ha) + (r_B + hb) = (r_A + r_B) + h(a + b) = r + hk$$

You thus obtain $r + hk$, a signature from the key k, even though participants used only the additive shares a and b in their computations. To obtain a threshold construction for arbitrary parameters t and n, Shamir's secret sharing can be used instead of additive sharing.

The previous construction isn't secure enough to satisfy all the security requirements of a threshold signing scheme. It's notably vulnerable to key cancellation attacks unless participants *commit* to their nonces in a preliminary phase of the protocol—for example, by sending the hash of their nonce. It also has a number of subtle vulnerabilities addressed by the protocol Flexible Round-Optimized Schnorr Threshold (FROST) signatures, designed by cryptographers Chelsea Komlo and Ian Goldberg in 2020 and documented at *https://eprint.iacr.org/2020/852*.

NOTE *The EdDSA signature protocol from Chapter 12 is similar to Schnorr's signatures but computes the nonce as a hash from the message, rather than as a random, arbitrary value. This complicates the creation of threshold signature protocols compliant with the original EdDSA specification.*

The Hard Case

The hardest signature scheme to run in a threshold setting is also the most common. The ECDSA signature algorithm (see Chapter 12) is more complex than Schnorr signatures and EdDSA because it involves a division. Given a message hash $h = \mathbf{H}(M)$, the signer picks a random number k, computes the number r from the point kG's coordinates, and computes the signature as $s = (h + ra) / k$, where a is the private key of the signer.

Efficient and secure ECDSA threshold signatures remain a challenging research problem among cryptographers. The first practical protocols appeared in the late 2010s, motivated by the use case of cryptocurrency—most of the leading cryptocurrencies, including Bitcoin and Ethereum, then supported only ECDSA as a transaction signing scheme.

Cryptographers devised several approaches to build ECDSA threshold signing protocols. For example, Yehuda Lindell's 2017 "Fast Secure Two-Party ECDSA Signing" (*https://eprint.iacr.org/2017/552*) required a commitment scheme, homomorphic encryption scheme, and zero-knowledge proof system. The complexity of such protocols complicated their understanding, implementation, and security analysis, leading to a number of security flaws in deployed systems.

How Things Can Go Wrong

Specific security issues in threshold signature protocols are often complex and involve details of cryptographic constructions that I haven't covered in this book. Therefore, instead of specific problems, I'll discuss categories

of problems that impacted real deployments of threshold signatures, from widely used open source software to commercial solutions.

Papers vs. Code

When engineers need to implement a threshold signature protocol from their research papers, they encounter challenges. These papers, primarily aimed at cryptography researchers, often vary in editorial quality and are complicated, heavy in mathematics, and quite new. This recency can mean that an experimental protocol might not be as secure as intended. These factors contribute to a range of real-world security issues, which can be categorized into four main areas:

Insecure protocol If the protocol isn't secure on paper, it won't be more secure in implementation. Common issues include overlooked edge cases or inadequate validation when receiving inputs from other parties. For example, some protocols failed to verify if an encrypted number falls within the expected range, leading to practical attacks.

Incomplete description Research papers aren't technical specifications but are written for academic audiences, so they often lack practical implementation details like networking and encoding. A notable example is the TSSHOCK attack on threshold signatures, which exploited ambiguous encoding in a hash function's input elements, as described in "Collisions from Domain Separation Failures" on page 305.

Incomplete implementation The complexity of threshold signing protocols, with their numerous subcomponents and detailed requirements, can lead to overlooked security validations, particularly if these are mentioned only in appendixes. An instance of this was when a protocol required a number $N = pq$ (for Paillier encryption) to be verified as the product of two large enough primes via a zero-knowledge proof, but this verification was omitted in the implementation, allowing insecure N values.

Insecure component choice Descriptions of protocols usually don't say "use the hash function BLAKE3" or "use the 256-bit elliptic curve nistp256"; instead, they say "use a hash function and an elliptic curve that offer the security level you need." It's thus up to the implementers to pick suitable primitives and use their programming interfaces securely. For example, using an RSA modulus of 1,024 bits isn't enough to ensure 128-bit security.

Additionally, risks arise when implementers intentionally modify a protocol, perhaps to enhance efficiency or fit a specific use case. This usually doesn't end well. For examples of attacks on threshold signatures, see the papers "Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations" by Dmytro Tymokhanov and Omer Shlomovits (*https://eprint.iacr.org/2021/1621*) and "Practical Key-Extraction Attacks in Leading

MPC Wallets" by Nikolaos Makriyannis, Oren Yomtov, and Arik Galansky (*https://eprint.iacr.org/2023/1234*).

Key Management Aspects

A friend once remarked, "For every 10 lines of encryption code, there are 1,500 lines of key management," which emphasizes the criticality and intricacy of key management processes, including key creation, storage, backup, and recovery. While these facets may be overlooked in academic papers focusing on theoretical aspects, they're vital in practical applications. Engineers and security professionals implementing threshold signing in a production environment must prioritize these key management issues—even the most robust threshold signature protocols are no substitute for comprehensive key management practices.

Let's review the main key management considerations for an organization employing threshold signatures to safeguard substantial cryptocurrency holdings:

Key generation Whether you're using distributed key generation or centralized generation, you must ensure that the secret values aren't exposed to unauthorized systems or parties during or after the key generation. This assurance is generally provided through rigorous processes such as key ceremonies, ensuring supply-chain integrity and audit trail, to demonstrate that keys have been generated properly. Don't allow another party (such as a cloud provider) to generate keys on your behalf or to have the capability to read them at any time.

Key storage Storing a key as multiple shares instead of a single value doesn't diminish the necessity for secure storage. Safeguarding numerous secrets across diverse platforms can present more of a challenge than protecting a single secret on a unified platform. Key shares must be stored in some type of secure memory, protected against unauthorized access, tampering, and physical attacks.

Key backup and recovery Implementing a threshold scheme where, for instance, you require three out of six shares to sign a transaction, protects against the loss of key shares or system outages. Nevertheless, this doesn't negate the necessity for key backups, which you should also maintain as threshold shares. It's imperative to distribute access to these backup shares among distinct parties. Additionally, validate the reliability of backups periodically to ensure they haven't been compromised and can be effectively utilized to reconstruct the key when necessary, such as during disaster recovery drills.

Zero-Knowledge Proofs

Zero-knowledge proofs (ZKPs) are one of cryptographers' most powerful tools. These are protocols between two parties, a *prover* and a *verifier*, where the prover convinces the verifier that something is true without revealing

anything about why. For example, the prover could prove that they know a solution to some hard computational problem without revealing the solution. You might create a ZKP for any **NP**-complete problems to prove you know a solution without revealing it.

More generally, ZKPs are used to prove a statement, such as "the number encrypted in this ciphertext is between 100 and 200" or "for a given plaintext P and ciphertext C, I know a secret key K for which $C = \mathbf{AES}(K, P)$."

For a non-technical introduction to zero-knowledge proofs, I recommend cryptographer Amit Sahai's video "Computer Scientist Explains One Concept in 5 Levels of Difficulty" (*https://youtu.be/fOGdb1CTu5c*). For mathematical details, see the links on *https://github.com/matter-labs/awesome-zero-knowledge-proofs*.

NOTE *The term* zero-knowledge proof *is a simplification of more accurate terms from the research literature. For example, many "zero-knowledge proof" protocols are in fact* zero-knowledge arguments of knowledge. *Researchers reserve the term* proof *for unconditional security and use* argument *for computational security. Furthermore, the term* witness *refers to the secret allowing the prover to prove its statement, as a generalization of* secret *or* secret key*, due to its broader scope.*

Security Model

What does it mean for a ZKP protocol to be secure? Is the attacker the prover or the verifier? How can they attack the protocol? Let's answer these questions by examining security goals and attacker models.

Security Goals

A secure ZKP must satisfy the following notions:

Completeness If the prover follows the protocol using the correct secret, an honest verifier is convinced of the truth of the statement. In other words, the protocol always works.

Soundness If the prover doesn't know the secret, they can't convince the honest verifier of a false statement (except with negligible probability). In other words, provers can't cheat.

Zero-knowledge A verifier learns nothing beyond the fact that the statement is true. Specifically, they can't learn anything about the prover's secret.

The notion that a verifier "will be convinced of the truth of the statement" is guaranteed by the mutual agreement that the protocol satisfies these three properties—namely, that a prover could not complete the protocol if the statement was false (for example, if they didn't know the solution they claim to).

Attacker Models

Both parties can potentially be attackers attempting to compromise soundness and zero-knowledge, respectively:

Malicious provers Want to prove a false statement—for example, to wrongly convince the verifier that they know a solution to some hard problem. Such an attacker can deviate from the protocol to try to fool the verifier. In practice, this is the greatest threat to ZKPs in applications like confidential program execution.

Malicious verifiers Want to extract information about the secret (the *witness*), thereby breaking the zero-knowledge property. Attacker models distinguish *passive* verifier attackers (honest-but-curious) and *active* verifier attackers (who can arbitrarily deviate from the protocol).

Note that malicious verifiers can challenge completeness by claiming they aren't convinced by the truth of a statement. In practice, this isn't an issue, as the prover can repeat the proof protocol for other (honest) verifiers who would expose the lying verifier.

Schnorr's Protocol

Claus-Peter Schnorr, who created the eponymous signature scheme, also described a similar construction that is a zero-knowledge *proof of knowledge of a discrete logarithm*. It's the basis for Schnorr's and EdDSA signatures, as well as for many more complex ZKPs.

Schnorr's protocol works in three steps to prove the knowledge of a such that $aG = A$—that is, the discrete logarithm of A with respect to the generator G:

1. *Commitment*: The prover picks a random number r and sends $R = rG$.
2. *Challenge*: The verifier sends a random number c.
3. *Response*: The prover sends $s = r + ca$, and the verifier accepts if and only if $sG = R + cA$.

NOTE *This kind of three-step protocol, with a commitment, a challenge, and a response, is called a* sigma protocol*, after the shape of the uppercase Greek letter sigma (Σ).*

The *completeness* of Schnorr's protocol is the easiest to verify: if a satisfies $aG = A$, you'll have

$$sG = (r + ca)\,G = rG + caG = R + cA$$

which is the prover's validation condition.

To see that the protocol ensures *soundness*—that the prover must know a—imagine that the prover uses the same r twice in two runs of the

protocol. The verifier thus gets two responses, $s_1 = r + c_1 a$ and $s_2 = r + c_2 a$, for the two distinct challengers c_1 and c_2. Now they can compute

$$s_1 - s_2 = r + c_1 a - r - c_2 a = a(c_1 - c_2)$$

and divide the result by $(c_1 - c_2)$ to obtain a. Since the check that sG equals $R + cA$ ensures that you had $s = r + ca$, it follows that the prover must know a when executing this protocol correctly. This kind of logical reasoning is called a *knowledge extractor*, and it's the main technique to show that a ZKP is sound.

The protocol can also be proved to be zero-knowledge, using a technique called a *simulator*, an algorithm that creates messages (or a *transcript of communication*) that are indistinguishable from those in a real execution of a zero-knowledge proof. However, unlike a real prover, the simulator doesn't necessarily know the secret (or witness) being proven. Despite this, the messages it generates still appear valid and convincing to the verifier in the context of the proof system.

In Schnorr's protocol, the simulator works backward and first chooses a random response, s, to show that a real s will be "as random" as a purely random one. Then it picks a random challenge c and computes the original commitment as $R = sG - cA$. The proof that the protocol is zero-knowledge then demonstrates that these three values are indistinguishable from those of a real execution of a protocol yet don't require the knowledge of the secret a. (Note that in Schnorr's case, you must assume that the verifier follows the protocol and picks a random c.)

Noninteractive Proofs

Schnorr's protocol is *interactive*: the prover sends a first message, the verifier responds with a challenge, and the prover sends them a response—the parties interact over three rounds of messages. But what if the verifier can't send messages and just wants to receive a single message that convinces them? How can one create such *noninteractive* proofs from an interactive protocol?

Let's take another look at the Schnorr protocol, wherein the verifier sends a random challenge c, a value that must be unpredictable to the prover. If the prover knows c before sending R, they can cheat, as follows: given c, choose some arbitrary value for s, and compute $R = sG - cA$; then send this R to the verifier, and send s as a response to c. Verification succeeds, yet the prover didn't need to use the secret value a.

How do you make c unpredictable to the prover without interacting with a verifier? The trick is to derive c from R using a hash function, which prevents the prover from finding a pair (R, c) that satisfies $sG = R + cA$ due to the pseudorandom behavior of hash functions.

To generate a *noninteractive zero-knowledge (NIZK)* proof of knowledge of the secret a using Schnorr's protocol, a prover proceeds as follows:

1. *Commitment*: The prover picks a random number r and computes $R = rG$.
2. *Challenge*: The prover computes $c = \mathbf{H}(G \,||\, R \,||\, A)$; you must include the values G and A to bind the c generation to the generator parameter G and to the public key A of the prover.
3. *Response*: The prover computes $s = r + ca$ and generates the proof as an encoding of R and s.

To verify a proof (R, s) received from the prover with public key A, a verifier recomputes the challenge $c = \mathbf{H}(G \,||\, R \,||\, A)$ and checks the equality if $sG = R + cA$.

The trick in hash function protocol data to replace verifier-generated challenges was formalized by the *Fiat–Shamir transform*, a general technique to turn an interactive protocol into a noninteractive one. For this transform to be applicable, a verifier's random challenges must be independent of the prover's message and be public (nonsecret) values.

zkSNARKs

Let's talk about the kind of zero-knowledge proof that's seen wide adoption in blockchain applications due to its power and efficiency. For example, zkSNARKs are the cornerstone of the Zcash confidential transaction platform: in Zcash, zkSNARKs prove that a certain amount has been deducted from an account and credited to another account, without revealing the amount of the accounts and without leading to a prohibitively large amount of computation or storage.

A *zkSNARK* is a type of noninteractive proof of knowledge that offers the zero-knowledge (*zk*) property, where *SNARK* stands for the following:

Succinct The proof is very small compared to the size of the statement and the secret. It may be similar in size to the statement size's logarithm, or even of *constant size*—always of the same size regardless of the statement's size.

Noninteractive A SNARK is a noninteractive argument of knowledge, typically using the Fiat–Shamir transform to turn an interactive protocol into a noninteractive one. It doesn't need the verifier to send messages to the prover.

Argument The *argument* of knowledge is a computationally secure proof that is conditionally secure. In other words, it wouldn't be secure against an attacker with infinite computing power, which is generally a tolerable limitation.

Of knowledge A SNARK offers completeness and soundness, as an argument of knowledge.

In addition, proving and verifying a zkSNARK must be computationally efficient.

Generating such a succinct proof when you want to prove the knowledge of a solution to a problem whose description doesn't even fit in the proof size sounds counterintuitive. For example, what about statements like

"I know a solution to the equation $f(x) = 0$," when the equation $f(x)$ can be of arbitrary size? From a theoretical perspective, short proofs make sense because the only information a proof must convey is the *knowledge* of some solution and generally the *correctness* of some statement, as opposed to the actual solution and secrets; the proof must be zero-knowledge and reveal only that information to the verifier.

In 2016, cryptographer Jens Groth published the article "On the Size of Pairing-Based Non-interactive Arguments" (*https://eprint.iacr.org/2016/260*), which describes an exceptionally efficient zkSNARK. The proof consisted of only three group elements and could be verified by computing three pairing operations (the same type of pairing as with BLS signatures). The Zcash protocol adopted this breakthrough result, which laid the foundation for several other zkSNARKs. Groth's zkSNARK is usually just called *Groth16*.

From Statements to Proofs

zkSNARKs are some of the most complex cryptographic constructions, with one of their most complex and costly steps being *arithmetization*, an operation that converts the statement to prove into a fixed number of polynomial equations, which are generally of the form

$$a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n$$

for a polynomial of degree n, where the coefficients a_i are elements of some finite ring or finite field structure. The proving algorithm then processes the polynomials to create the zkSNARK proof.

Arithmetization follows a general workflow:

1. Describe the statement to prove using formal notations, such as a computer program, equations, or logical formulas.
2. Transform the formal expression from step 1 into a *circuit*, which defines an output from input values by applying a sequence of *gates* to the input, similar to logical gates in a Boolean or electronic circuit, except that the gates may be algebraic operations such as addition and multiplication.
3. Turn the circuit into a structured list of *constraints*, according to the zkSNARK proof system's constraint system. Such constraints are lists of conditions that the input must fulfill to satisfy the statement to prove.

The statement to prove may be as trivial as "I know integer x and y satisfying the equation $x^3 + y^2 + xy + 55 = 0 \bmod 57$." This completes step 1 by having a formal equation expressing the problem. To complete step 2, you may break down the equation as a sequence of simple operations involving two operands (Groth16 requires this). This works as follows, where you write the intermediate values $v_0, v_1, \ldots, v_6$:

Set $v_0 = x \times x$.

Set $v_1 = x \times v_0$; thus $v_1 = x^3$.

Set $v_2 = y \times y$.

Set $v_3 = x \times y$.

Set $v_4 = v_1 + v_2$.

Set $v_5 = v_4 + v_3$.

Set $v_6 = v_5 + 55$; thus $v_6 = x^3 + y^2 + xy + 55$.

Such a translation of a long equation into a series of small ones is called *flattening*.

The prover then converts these operations—the circuit—into a set of mathematical structures that serve to construct the polynomials processed by the prover. The long polynomials are finally "compressed" into a proof using randomness, specifically the concept of *probabilistically checkable proof (PCP)*, which is a major discovery from the field of complexity theory. This convinces a verifier that many constraints are satisfied with only a few actual constraint checks, while preserving the zero-knowledge property.

To learn more about the intricacies of arithmetization, research the two main approaches used by zkSNARKs: rank-1 constraint systems (R1CS) and algebraic intermediate representations (AIR).

Finally, note that we distinguish *nonuniversal* and *universal* zkSNARK proof systems: in the former, the prover works for a specific, predefined statement. In particular, the setup phase of the proof system creates parameters suitable only to a given statement. However, universal proof systems such as Marlin and Plonk take as input a statement and create a proof for it. They're more flexible but also more complex to construct and have higher computation overhead.

How Things Can Go Wrong

zkSNARKs can suffer from the same classes of problems as in the context of threshold signatures, from an insecure protocol to implementation flaws. Security issues can arise at different stages of the workflow, from the statement definition to the arithmetization step and the proof computation. The security notion impacted may be completeness, soundness, or zero-knowledge. But most of the time, the greatest risk concerns soundness, or the possibility of an attacker to cheat and fool a verifier—both because of the potential impact on the application and the subtlety of soundness bugs, whereas a leak of "knowledge" is less likely, especially if the proof must remain a valid one accepted by a verifier.

In the forthcoming examples, we'll focus on Schnorr's protocol. The problems are relatively simple, but more complex proof systems can have much subtler issues.

Recall that Schnorr's noninteractive protocol proves the knowledge of a to a verifier that knows $A = aG$ by sending a verifier $s = r + ca$ and $R = rG$ for a random r and $c = \mathbf{H}(G \,\|\, R \,\|\, A)$. The verifier then checks the equality $sG = R + cA$ after recomputing c. In the interactive version, the verifier chooses c randomly.

Insufficient Fiat–Shamir Hashing

Imagine that instead of $c = \mathbf{H}(G \,||\, R \,||\, A)$, a noninteractive Schnorr proof did $c = \mathbf{H}(G \,||\, A)$, making the challenge c independent of the nonce R. The attacker then picks an arbitrary value for s and computes $R = sG - cA$. The resulting proof consisting of s and R is valid, yet the attacker didn't need to know a, thereby breaking the protocol's soundness.

Likewise, if $c = \mathbf{H}(G \,||\, R)$, omitting the public key A from the data hashed to define the challenge c, an attack is possible: the attacker can now choose arbitrary R and s values and compute the point $B = (1/c) \times (sG - R)$, which satisfies $sG = R + cB$. This gives the attacker a proof of knowledge of the discrete logarithm of B—namely, b such that $B = bG$—whereas they don't know it.

These attacks illustrate the importance of including all necessary values in the hash function input when using the Fiat–Shamir transform to make a protocol noninteractive.

Replay Attacks

A replay is a trivial but potentially devastating attack. If an attacker learns the value of some noninteractive proof of knowledge, they can send it to another party and say they created it, claiming the credit for the knowledge proven.

You may avoid this by binding the proof to the identity of the prover, including their public key in the data hashed. To prevent replay by the same party over time, you may bind the proof to a session identifier or timestamp by including such values in the data processed by the Fiat–Shamir hashes.

Randomness Reuse

Consider the interactive Schnorr protocol, where the verifier picks a random c. If the prover has a flawed pseudorandom generator and reuses the same challenge r twice, then an attacker observing the exchanged values can recover the secret a by using the two proofs $s_1 = r + c_1 a$ and $s_2 = r + c_2 a$ for two distinct challengers, c_1 and c_2, and computing $a = (s_1 - s_2) \,/\, (c_1 - c_2)$.

Really Serious Crypto

With this final chapter, we've reviewed some of the most captivating topics in cryptography at the time of writing, from both theoretical and practical viewpoints. We've touched only the surface, though, particularly in the realm of zero-knowledge proof systems, an active research and engineering space with extensive applications beyond blockchain technology. But cool cryptography isn't the panacea of blockchains. From multiparty computation protocols such as private set intersection (PSI) to homomorphic encryption used for private evaluation of AI models, new applications and use cases call for better, faster cryptographic functionalities.

We're witnessing a golden age in cryptography, with an unprecedented convergence of theoretical principles and practical applications. This

synergy is providing almost magical solutions to some of the most daunting security and privacy problems. Nonetheless, significant challenges need to be addressed, particularly in the legal and regulatory spheres. It's imperative that technologists and policymakers collaborate closely to navigate these challenges and that both attempt to understand each other's perspectives. It is my hope that this book, and particularly this last chapter, contributes to demystifying cryptography and making it more accessible and less enigmatic for all its readers.

INDEX

Numbers

0-RTT data, 267
3DES (triple DES), 67, 82–83. *See also* DES

A

A5/1, 21, 98–101
Aaronson, Scott, 185, 193, 281, 293
active attacker, 318
adaptive corruption, 318
Advanced Encryption Standard (AES), 61, 67
  AddRoundKey, 68–70
  block size, 62
  vs. DES, 67, 90
  and GCM, 164–167, 171, 173
  implementations, 71–74
  internals, 67–70
  KeyExpansion, 68–69
  MixColumns, 68–69
  with Poly1305, 148
  and provable security, 50
  security of, 73
  ShiftRows, 68–69
  SubBytes, 68–69
  and TLS 1.3, 265–266
Advanced Vector Extensions (AVX), 63
AEAD (authenticated encryption with associated data), 18, 160, 169–170
AES. *See* Advanced Encryption Standard
AES-CBC, 78
`AESENC` instruction, 72
`AESENCLAST` instruction, 73
AES-GCM
  efficiency, 166
  internals, 164–165
  security, 166
  and small tags, 173
  and weak hash keys, 171–173
AES native instructions (AES-NI), 72
AEZ, 174
aggregate signature protocols, 311–314
AKA (authenticated key agreement), 220–221
algebraic attacks, 96
Alvisi, Lorenzo, 136
amplitude, 274–279
Apple, 243, 246
application-specific integrated circuit (ASIC), 89
arithmetization, 328
associated data, 160
asymmetric encryption, 3, 18. *See also* RSA
attack costs, 47–49
attack models, 12
  black-box, 13–14
  gray-box, 14
  for key agreement protocols, 221
authenticated ciphers, 160
  with associated data, 160–161
  functional criteria, 163
  nonces, 161–162
  online, 163
  performance, 162–163
  permutation-based, 169–171
  security, 162
  streamability, 163
authenticated decryption, 160
authenticated Diffie–Hellman, 224–226
authenticated encryption (AE), 18, 157
  AES-GCM, 164–167, 171–173
  authenticated ciphers, 160–163
  OCB, 167–169
  permutation-based AEAD, 169–171
  SIV, 169
  using MACs, 158–160

authenticated encryption with associated data (AEAD), 18, 160, 169–170
authenticated key agreement (AKA), 220–221
authentication tag, 18. *See also* authenticated encryption; MACs
AVX (Advanced Vector Extensions), 63

B

backtracking resistance, 30
backward secrecy, 30
`BcryptGenRandom()` function, 37–38
Bellare, Mihir, 155
Bellaso, Giovan Battista, 5
Bellcore attack, 211
Bernstein, Daniel J., 57, 106, 110, 148, 151, 244, 248, 283
big-number libraries, 206
bilinearity, 312
binary exponentiation, 207
birthday attacks, 120
birthday paradox, 120
Bitcoin, 116, 297, 299–302, 306, 307
bit security, 46–48
BLAKE, 131
BLAKE2, 115, 133–135, 143
  BLAKE2b, 134
  BLAKE2s, 134
  compression function, 134–135
  design rationale, 134
blinding attacks, 203
block ciphers, 61. *See also* Advanced Encryption Standard
  block size, 62–63
  CBC mode, 76–78
  codebook attacks, 63
  CTR mode, 80–82
  decryption algorithm, 62
  ECB mode, 74–76
  encryption algorithm, 62
  Feistel schemes, 66–67
  key schedule, 64
  meet-in-the-middle attacks, 82–83
  modes of operation, 74
  padding oracle attacks, 83–85
  round keys, 64–65
  rounds, 64
  security goals, 62
  slide attacks, 64–65
  substitution–permutation networks, 65–66
BLS curve, 312
BLS signature, 312–313
Bluetooth, 88
Boneh, Dan, 214
Bos, Joppe W., 251
broadcast attack model, 105
Brumley, David, 214
brute-force attacks, 45, 100–101

C

CA (certificate authority), 258–262, 269–270
cache-timing attacks, 72
Caesar cipher, 4–5
CAESAR competition, 174
Canetti, Ran, 155
carry-less multiplication (`CLMUL`), 165
CBC. *See* cipher block chaining
CBC-MAC, 146–147
CCA (chosen-ciphertext attackers), 13–14
CCM (counter with CBC-MAC), 174, 265
CDH (computational Diffie–Hellman), 218–219
certificate authority (CA), 258–262, 269–270
certificate chain, 259, 269
ChaCha20, 106, 131, 151, 265
chaining values, 122
Chinese remainder theorem (CRT), 210–211
chosen-ciphertext attackers (CCA), 13–14
chosen-message attacks, 140
chosen-plaintext attackers (CPA), 13
Chrome browser, 129, 271
Chuang, Isaac, 293
cipher-based MAC (CMAC), 146–147
cipher block chaining (CBC), 76–78
  ciphertext stealing, 79
  padding, 78–79
  padding oracle attacks, 83–85

ciphers, 3
ciphertext, 4
ciphertext-only attackers (COAs), 13
ciphertext stealing, 79
circuit, 328
C language, 91
Clay Mathematics Institute, 50, 185
client certificate, 268
clique problem, 184
CLMUL (carry-less multiplication), 165
closest vector problem (CVP), 287
CMAC (cipher-based MAC), 146–147
CMAC-AES, 169
code-based cryptography, 285–286
codebook attacks, 63, 101
Codenomicon, 270
coding problems, 194
Cohen, Henri, 251
Cold War, 61
collision resistance, 119–120, 123, 305–306
completeness, 324
complexity. *See* computational complexity
complexity class, 182
complex numbers, 275
compression functions, 122
  in BLAKE2, 134–135
  Davies–Meyer construction, 124–125
  in Merkle–Damgård construction, 122
  in SHA-1, 127
computational complexity, 178
  bounds, 182
  classes, 182
  comparison, 180
  constant factors, 179
  constant time, 179
  exponential, 179–181
  exponential factorial, 181
  linear, 179
  linearithmic, 179
  polynomial, 180–183
  quadratic, 180
  superpolynomial, 180
computational complexity theory, 177
computational Diffie–Hellman (CDH), 218–219
computational hardness, 178
computational security, 44–46
confidentiality, 3, 21, 116, 162
confusion, 65, 303
constant-time implementations, 154
Coppersmith, Don, 213
counter mode (CTR), 80–82, 102, 164
counter with CBC-MAC (CCM), 174, 265
CPA (chosen-plaintext attackers), 13
CRCs (cyclic redundancy checks), 116
CRT (Chinese remainder theorem), 210–211
CryptAcquireContext() function, 34
CryptGenRandom() function, 37
Crypto++, 214
Cryptocat, 41
cryptographic security, 43. *See also* security
CTR (counter mode), 80–82, 102, 164
cube attacks, 96
Curve448, 265
Curve25519, 248, 265
Curve41417, 248
CVP (closest vector problem), 287
cyclic redundancy checks (CRCs), 116

D

Dahlin, Mike, 136
Damgård, Ivan, 122, 237
Data Encryption Standard. *See* DES
Datagram Transport Layer Security (DTLS), 257
Davies–Meyer construction, 124–125, 127, 134
decisional Diffie–Hellman (DDH)
  assumption, 219
  problem, 218–219
dedicated hardware, 90
DeMillo, Richard A., 214
DES (Data Encryption Standard), 61, 90
  vs. AES, 67, 90
  block size, 62
  double DES, 83
  Feistel schemes in, 66–67
  3DES, 67, 82–83
deterministic random bit generator (DRBG), 16, 30, 88

/dev/random, 36–37
/dev/urandom, 34–37
Diehard, 33
differential cryptanalysis, 109
Diffie, Whitfield, 195, 215
Diffie–Hellman problem, 194
Diffie–Hellman protocol, 241
  anonymous, 223–224
  authenticated, 224–227
  CDH problem, 218
  DDH problem, 218–219
  function, 216–217
  generating parameters, 217
  and key agreement, 219–222, 241
  MQV protocol, 227–228
  and shared secrets, 216, 228–229
  in TLS, 215, 229, 264–265
  twin problem, 219
  unsafe group parameters, 229–230
diffusion, 65–66, 304
digest, 116
DigiNotar, 269
digital signatures, 116–117, 196, 202–205
discrete logarithm problem (DLP), 189–191
  and CDH problem, 218
  ECDLP, 240–241
  and Shor's algorithm, 281–282
dishonest majority, 318
distribution, 26–27
domain separation, 305–306
`drand48`, 32
DRBG (deterministic random bit generator), 16, 30, 88
DTLS (Datagram Transport Layer Security), 257
Durumeric, Zakir, 40

E

ECB (electronic codebook), 74
ECC (elliptic curve cryptography), 231
ECDH (elliptic curve Diffie–Hellman), 241, 249, 292
ECDLP (elliptic curve discrete logarithm problem), 240–241
ECDSA. *See* elliptic curve digital signature algorithm
ECIES (elliptic curve integrated encryption scheme), 246
Ed25519, 244, 250
Ed448-Goldilocks, 248
EdDSA, 244
Einstein–Podolsky–Rosen (EPR) paradox, 274
elliptic curve cryptography (ECC), 231
elliptic curve Diffie–Hellman (ECDH), 241, 249, 292
elliptic curve digital signature algorithm (ECDSA), 241, 321
  and bad randomness, 249
  vs. RSA signatures, 243–244
  signature generation, 242
  signature verification, 242–243
elliptic curve discrete logarithm problem (ECDLP), 240–241
elliptic curve integrated encryption scheme (ECIES), 246
elliptic curves, 231
  addition law, 235
  Curve25519, 248, 265
  Curve41417, 248
  Curve448, 265
  Edwards curves, 233, 244
  groups, 239–240
  with integers, 233–234
  NIST curves, 247–248
  order, 240
  pairing, 312
  point at infinity, 236, 239–240, 314
  point doubling, 237–238
  point multiplying, 238–239
  prime curves, 247
  Weierstrass form, 232
embarrassingly parallel, 48, 100
Encapsulating Security Payload (ESP), 263
encrypt-and-MAC, 158–159
encryption, 3
  asymmetric, 18
  at-rest, 17
  in-transit, 17
  randomized, 15–16
  security, 12
encrypt-then-MAC, 158, 159–160, 164
entanglement, 274, 277

entropy, 27–31, 36–37
entropy pool, 29–31
EPR (Einstein–Podolsky–Rosen) paradox, 274
error-correcting codes, 285
ESP (Encapsulating Security Payload), 263
eSTREAM competition, 97, 106, 113
Ethereum, 306
*e*th roots, 199
Euler's theorem, 212
Euler's totient function, 196
exponentiation, 206–208, 210
extended Euclidean algorithm, 198

F

factorials, 9
factoring methods, 187
factoring problem, 50–51, 186
  and **NP**-completeness, 188–189
  solving with Shor's algorithm, 281–282
factorization, 187–188, 191–192
fast correlation attacks, 96
fault injection, 211
FDH (Full Domain Hash), 205
feedback shift registers (FSRs), 90–92
  cycle, 92
  feedback function, 90
  linear, 93–95
  nonlinear, 96
  period, 92
Feistel schemes, 66–67
Ferguson, Niels, 30, 173
FHE (fully homomorphic encryption), 20
Fiat–Shamir transform, 327, 330
field-programmable gate array (FPGA), 89
filtered LFSR, 95–96
first-preimage resistance, 118
fixed points, 125
Flame, 137
Flexible Round-Optimized Schnorr Threshold (FROST), 321
forgery attacks, 140
format-preserving encryption (FPE), 19–20
Fortuna, 30–31
forward secrecy, 221–222, 225, 228
  in authenticated DH, 225
  in TLS 1.3, 268–269
Fouque, Pierre-Alain, 155
FOX, 66
FPGA (field-programmable gate array), 89
frequency analysis, 6
Frey, Gerhard, 251
FROST (Flexible Round-Optimized Schnorr Threshold), 321
FSRs. *See* feedback shift registers
full diffusion, 110
Full Domain Hash (FDH), 205
fully homomorphic encryption (FHE), 20

G

GCD (greatest common divisor), 40, 198, 211, 282
GCHQ (Government Communications Headquarters), 216
GCM (Galois Counter Mode), 158, 164, 173. *See also* AES-GCM
`gcm_ghash_clmul` function, 165
general number field sieve (GNFS), 187, 218
`getrandom()` system call, 36
GHASH, 165–166, 171–172
Gilbert, Edgar, 148
Git, 115
GitHub, 55
Gmail, 268, 270
GMR-1, 113
GMR-2, 113
GNFS (general number field sieve), 187, 218
GNU Multiple Precision (GMP), 206
Go, 152, 206, 208
Goldberg, Ian, 39, 321
Goldwasser, Shafi, 22
Google, 129, 258–261, 269, 270
  Chrome browser, 129, 271
GOST, 61, 67
Govaerts, René, 137
Government Communications Headquarters (GCHQ), 216

Grain-128a, 97–98
graphics processing unit (GPU), 101, 178
greatest common divisor (GCD), 40, 198, 211, 282
Grøstl, 131
groups
  axioms, 190
  commutativity, 190
  cyclic, 190
  finite, 190
  generator, 190
  in RSA, 196–197
Grover's algorithm, 282–283
GSM mobile communication, 88, 140
guess-and-determine attacks, 100–101

H

Hadamard gate, 278–279
Halderman, Alex, 40, 251
hardness assumption, 189
hard problems, 177. *See also* computational complexity
  closest vector problem, 287
  discrete logarithm problem, 189–191
  factoring problem, 50–51, 186
  learning with errors, 194, 286
  multivariate quadratic equations, 287
  NP-complete problem, 183–186
  P vs. **NP** problem, 185–186
  and provable security, 50–51
  short integer solution, 194, 286
hardware, 72, 113
hash-based cryptography, 288–289
hash-based MACs, 144–145
hash functions, 115. *See also* Merkle–Damgård construction
  collisions in, 119–121
  compression functions, 122
  Davies–Meyers construction, 124–125
  in digital signatures, 116
  iterative, 122
  keyed, 139
  multicollisions, 123–124
  noncryptographic, 116
  P vs. **NP** problem, 185–186
  preimage resistance, 117–119
  in proof-of-storage protocols, 136–137
  security notions, 116
  sponge functions, 122, 125–126
  3-collisions, 123
  universal, 148–149
  unpredictability, 117
hash values, 116, 117–121
Heartbleed, 256, 270
Hellman, Martin, 195, 215
Heninger, Nadia, 40, 251
heuristic security, 50, 52–53
HMAC-based KDF (HKDF), 229, 265
HMACs (hash-based MACs), 144–146
honest majority, 318
HTTPS, 258
  insecure, 166, 193
  keys for, 53, 56
  over TLS, 104, 215, 256

I

iCloud, 270
identity gate, 278
IES (integrated encryption scheme), 246
IETF (Internet Engineering Task Force), 134, 164, 250, 314
IKE (Internet Key Exchange), 146
imaginary number, 275
IND-CPA, 15–17
indifferentiability, 137
indistinguishability (IND), 15
initial value (IV), 76, 81, 89, 122, 147
integrated encryption scheme (IES), 246
integrity, of data, 19, 116, 140
Intel, 38, 73, 165
Internet Engineering Task Force (IETF), 134, 164, 250, 314
Internet Key Exchange (IKE), 146
internet of things (IoT), 255
intractable problems. *See* hard problems
invalid curve attack, 249–250
invalid key, 314–315
invasive attacks, 14

ion traps, 284
ipad, 144
IPSec (Internet Protocol Security), 140, 144, 146, 160, 164, 263
iterative hashing, 122
IV (initial value), 76, 81, 89, 122, 147

J

Jager, Tibor, 250
Java, 22, 32, 152
JH, 131

K

KDF. *See* key derivation function
Keccak, 126, 131–133, 171. *See also* SHA-3
Kelsey, John, 30, 41, 50
Kerckhoffs, Auguste, 6, 13
Kerckhoffs's principle, 13
key agreement protocols, 53, 216, 219
  AKA, 220–221
  attack models, 221
  breaches, 221, 225, 228
  data leaks, 221, 226, 268
  eavesdroppers, 216, 218
  forward secrecy, 221–222, 225, 228
  performance, 222
  security goals, 221
key cancellation attack, 309–310, 315–316
key confirmation, 226, 228
key control, 221, 225
key derivation function (KDF), 53
  in DH functions, 216, 229
  in ECIES, 246
  in TLS 1.3, 265–266
key generation, 39, 40, 198, 323
key-generation algorithm, 53, 54
key management, 323
key scheduling algorithms (KSAs), 102–104
key wrapping, 55
knapsack problem, 184
knowledge extractor, 326
known-message attack, 140
known-plaintext attackers (KPAs), 13, 100
Knudsen, Lars, 51
Kohno, Tadayoshi, 31
Kotla, Ramakrishna, 136
Kozierok, Charles, 257
Krawczyk, Hugo, 155, 230
Krovetz, Ted, 168
KSAs (key scheduling algorithms), 102–104
Kupyna, 127

L

Lagrange interpolation, 319
lattice-based cryptography, 286–287
lattice problems, 194, 287, 291
learning with errors (LWE), 286, 291
least significant bit (LSB), 179, 207
length-extension attacks, 135–136, 142–143
Let's Encrypt, 271
Leurent, Gaëtan, 155
linear code, 285–286
linear combination, 33
linear feedback shift registers (LFSRs), 93
  in A5/1, 98–100
  filtered, 95–96
  in Grain-128a, 97–98
  polynomials, 93
  security, 95
linear transformation, 33, 93, 287
Linux, 34, 36, 74, 152
Lipton, Richard J., 214
logarithm, 27, 46
long-term key, 225
lower bound, 45
low-exponent attacks, 209
LSB (least significant bit), 179, 207
Lucifer, 66
LWE (learning with errors), 286, 291

M

MacBook, 154, 191, 209
MACs (message authentication codes), 139
  authentication tag, 140
  CBC-MAC, 146–147
  chosen-message attacks, 140
  CMAC, 146–147
  dedicated designs, 148

MACs *(continued)*
encrypt-and-MAC, 158–159
encrypt-then-MAC, 158, 159–160, 164
forgery attacks, 140
HMAC, 144–146
MAC-then-encrypt, 158–160
vs. PRFs, 141–142
replay attacks, 141
timing attacks, 140–142
Wegman–Carter, 149–150
MAC-then-encrypt, 158–160
MacWilliams, F.J., 148
malleability, 199–200
man-in-the-middle attacks, 220, 223–224, 256
mask generating function, 202
matrix multiplication, 278
McEliece cryptosystem, 285–286
MD5, 122, 127, 133
M–D construction. *See* Merkle–Damgård construction
measurement (quantum physics), 274, 278
MediaWiki, 40
meet-in-the-middle (MitM) attacks, 82–83
memory, 48
memory footprint, 63
Menezes–Qu–Vanstone (MQV), 227–228, 241
Merkle, Ralph, 122, 137, 216, 297
Merkle–Damgård (M–D) construction, 122
length-extension attacks, 135–136, 142–145
multicollisions, 123–124
padding, 123
security, 123
Merkle's puzzles, 216
Mersenne Twister (MT) algorithm, 32, 40, 305
message authentication codes. *See* MACs
Micali, Silvio, 22
Microsoft, 111, 131
Microsoft Windows CryptoAPI, 208
misuse resistance, 162
MitM (meet-in-the-middle) attacks, 82–83
mode of operation, 7–8, 61, 74
Moore, Jonathan, 251
most significant bit (MSB), 32, 147, 150, 229
MQ (multivariate quadratics), 287
MQV (Menezes–Qu–Vanstone), 227–228, 241
MT (Mersenne Twister) algorithm, 32, 40, 305
`mt_rand`, 32, 40
multicollisions, 123–124
multiparty computation (MPC), 316
multisignature protocols, 306
multivariate cryptography, 287–288
multivariate problems, 194
multivariate quadratics (MQ), 287
MuSig, 310–311

N

Naehrig, Michael, 251, 312
Netscape, 39, 257
network-based intrusion detection systems (NIDS), 115
Neves, Samuel, 133, 135
NFSR (nonlinear feedback shift register), 96–98
Nguyen, Phong Q., 155
Nielsen, Michael, 293
NIST (National Institute of Standards and Technology), 33, 61, 67, 129–131, 247–248, 289–291
NM (nonmalleability), 15
nonces, 80–82, 88–89
predictability, 161–162
reuse, 111, 169
in TLS records, 263
WEP insecurity, 103–104
nondeterministic polynomial time class. *See* **NP** class
noninteractive zero-knowledge (NIKZ), 326–327
nonlinear equation, 33, 96
nonlinear feedback shift register (NFSR), 96–98
nonmalleability (NM), 15
nonrepudiation, 202

nonuniform distribution, 27
NP (nondeterministic polynomial time) class, 182–183
NP-complete problem, 183–185
NP-hard problem, 185–186
NSA (National Security Agency), 10, 85, 105, 127, 129, 227, 244, 247, 273
NSS library, 214
number field sieve, 218

O

OAEP. *See* Optimal Asymmetric Encryption Padding
OCB (offset codebook)
efficiency, 168–169
internals, 167–168
security, 168
one-time pad, 9
encrypting with, 9–10
security, 10–11, 12, 44
one-way function, 117
opad, 144
OpenSSH, 148, 159, 231, 246, 248
OpenSSL toolkit
generating DH parameters, 217
generating keys, 53–54, 192–193
GHASH bug, 165
Heartbleed, 256, 270
unsafe DH group parameters, 229–230
Optimal Asymmetric Encryption Padding (OAEP), 56, 200
encoded message, 201
mask generating function, 202

P

P (polynomial time) class, 180–185
P vs. **NP**, 185–186
padding, 22, 78–79, 83–84, 123
OAEP, 56, 200
zero padding, 263
padding oracle attacks, 22, 83–84
pairing, 312
parallelism, 47–48
parallelizability, 162, 166, 168
parent process ID (PPID), 39
passive attacker, 318
password, 53, 55, 141
Peikert, Chris, 291
perfect secrecy, 9
period, 92–94, 97–98, 281–282
permutation, 6
permutation-based AEAD, 169–171
pseudorandom, 62, 150
security, 7, 8–9
in sponge functions, 125–126
trapdoor, 196, 197–199
PID (process ID), 39
pigeonhole principle, 119
PKCS (Public-Key Cryptography Standards), 200–201
plaintext, 4
PLD (programmable logic device), 89
Poly1305, 148–151
Poly1305-AES, 150–151
polynomials, 93
multiplication, 165–166
primitive, 93–94
polynomial time (**P**) class, 180–185
post-quantum cryptography, 274, 285
code-based, 285–286
hash-based, 288–289
lattice-based, 286–287
multivariate, 287–288
Post-Quantum Cryptography Standardization project, 289
post-quantum security, 283, 304
power-analysis attacks, 208
PPID (parent process ID), 39
PQCrypto, 293
precomputation, 48, 222
prediction resistance, 30
preimage resistance, 117–119
Preneel, Bart, 137
preshared key (PSK), 265, 267
PRFs. *See* pseudorandom functions
prime numbers, 187
prime number theorem, 187
private keys, 18, 195
PRNGs. *See* pseudorandom number generators
Probabilistic Signature Scheme (PSS), 203–205
probability, 11, 26

probability distribution, 26–27
process ID (PID), 39
programmable logic device (PLD), 89
proof-of-storage protocols, 136–137
proof-of-work, 300
provable security, 50–53
pseudorandom functions (PRFs), 139
    vs. MACs, 141–142
    security, 141
pseudorandom number generators (PRNGs), 28–30
    cryptographic, 32–33
    and entropy, 39–40
    Fortuna, 30–31
    generating on Unix, 34–36
    generating on Windows, 37–38
    hardware-based, 38
    noncryptographic, 32, 40
    security, 30
pseudorandom permutation (PRP), 62, 67, 150
PSK (preshared key), 265, 267
PSPACE, 182
PSS (Probabilistic Signature Scheme), 203–205
public-key cryptography, 18, 231
Public-Key Cryptography Standards (PKCS), 201
public-key cryptosystem, 215
public keys, 195
PyCrypto, 70
Pythagorean theorem, 275
Python language, 70, 75, 80–81, 102, 212

Q

Qualys, 271
quantum bit (qubit), 274, 276–279, 284
quantum byte, 277
quantum circuits, 278
quantum computers, 188–189, 274
quantum gates, 277–279
quantum mechanics, 274
quantum random number generators (QRNGs), 29
quantum speedup, 279
    exponential, 280
    quadratic, 280
quarter-round function, 106–107
qubit (quantum bit), 274, 276–279, 284

R

randomness, 25
random number generators (RNGs), 28–29
random oracle, 117
Ray, Marsh, 74
RC4, 89, 102
    broken implementation, 111–113
    in TLS, 104–105
    in WEP, 103–104
`RDRAND` instruction, 38
`RDSEED` instruction, 38
reduction, 50
replay attacks, 142, 220, 330
Rho method, 120–121
Rijndael, 67
ring-LWE, 291
Rivest, Ron, 102
Rivest–Shamir–Adleman. *See* RSA
RNGs (random number generators), 28–29
Rogaway, Phillip, 168, 169
rogue key attack, 310
root of unity, 212
rounds, 53
round trips, 222
round-trip times (RTTs), 267
RSA (Rivest–Shamir–Adleman), 195
    Bellcore attack, 211–212
    CRT, 210–211
    vs. ECDSA, 243–244
    encryption, 199
    and factoring problem, 50–51, 186
    FDH, 205
    groups, 196–197
    implementations, 205–206
    key generation, 208–209
    modulus, 212
    OAEP, 200–202
    private exponents, 212–213
    private keys, 54, 197, 198
    problem, 218
    PSS, 203–204, 205
    public exponents, 197
    public keys, 197

secret exponents, 197
security, 198
shared moduli, 212–213
signatures, 202–205
small exponents, 208–209
speed, 208–210
square-and-multiply, 207–208
textbook encryption, 199–200
textbook signature, 203
trapdoor permutation, 196, 197, 199
RSAES-OAEP, 200
RSA Security, 102
RTT (round-trip times), 267

S

Saarinen, Markku-Juhani O., 132, 173
safe prime, 217
SageMath, 191, 198, 239
Salsa20, 106
attacking, 110–111
column-round function, 108
double-round function, 107
internal state, 107
and nonlinear relations, 110
quarter-round function, 106–107
row-round function, 107
Salsa20/8, 110–111
salt, 204
sandwich MAC, 144
satellite phone (satphone), 113
S-boxes (substitution boxes), 65
scheduling problems, 184
Schneier, Bruce, 30, 31, 41, 131
Schnorr, Claus-Peter, 244, 307
Schnorr signature protocol, 307–309
Schnorr's proof-of-knowledge protocol, 325–326
Schwenk, Jörg, 250
searchable encryption, 20
search algorithm, 178
second-preimage resistance, 118
secret-prefix MAC, 143, 145
secret sharing, 319
additive, 319
threshold, 319
secret-suffix MAC, 143
secure channel, 216, 256
secure cookie, 268
Secure Hash Algorithm with Keccak (SHAKE), 132
Secure Shell (SSH), 55, 140, 144, 159, 160, 241, 262
Secure Socket Layer (SSL), 39, 255, 257
security
bit, 46–47
computational, 44–45
cryptographic, 43
goals, 12, 15
heuristic, 50, 52–53
levels, choosing, 49–50
margin, 53
notions, 12, 15–17
post-quantum, 283
proof, 46
provable, 50–52
semantic, 15, 16
session key, 53, 219
SHA-0, 127–128
SHA-1, 127–129, 266
SHA-2, 129–133
SHA-224, 129–130
SHA-256, 117, 123, 129–130, 242, 297, 300
compression function, 130, 146
security, 130–131, 297
SHA-3, 126, 132–133, 229
competition, 131–132
security, 133–134
Zoo, 137
SHA-384, 130
SHA-512, 130
SHAKE (Secure Hash Algorithm with Keccak), 132
Shamir's secret sharing, 319
Shannon, Claude, 10
SHAs (Secure Hash Algorithms), 126
Shor, Peter, 281
Shor's algorithm, 281–282
short integer solution (SIS), 286
Shrimpton, Tom, 169
side-channel attacks, 14, 65, 153, 292
Signal, 292
signatures, 116, 196, 202–205
SIM card, 220
Simon's problem, 280–281

Simple Mail Transfer Protocol (SMTP), 229, 257
simulator, 326
SipHash, 151–152, 155
SipRound function, 151–152
SIS (short integer solution), 286
SIV (synthetic IV), 169
Skein, 131
slide attacks, 64–65
sliding window method, 208
Sloane, N.J., 149
SM3, 127
SMTP (Simple Mail Transfer Protocol), 229, 257
SNOW3G, 102
Somorovsky, Juraj, 250
soundness, 324
space complexity, 182
SPHINCS+, 289–290
SPNs (substitution–permutation networks), 65–66, 68
sponge functions, 122, 125–126, 155
  absorbing phase, 126
  capacity, 126
  squeezing phase, 126
square-and-multiply, 207–208
SSH (Secure Shell), 55, 140, 144, 159, 160, 241, 262
SSL (Secure Socket Layer), 39, 255, 257
SSL Labs, 271
static corruption, 318
statistical test, 33–34
streamability, 163, 167–169
stream ciphers, 87
  counter-based, 89
  encryption and decryption, 88
  hardware-oriented, 89–90
  keystream, 88
  nonce reuse, 111
  software-oriented, 101
  stateful, 89
Streebog, 127
substitution boxes (S-boxes), 65
substitution–permutation networks (SPNs), 65–68
substitutions, 7
superconducting circuits, 284
superposition, 274
symmetric encryption, 3, 18
synthetic IV (SIV), 169

T

tags, 18. *See also* authenticated encryption; MACs
TEA, 137
threshold signature protocol, 316
time complexity, 180–182
time-memory trade-off (TMTO) attacks, 21, 48, 101
timing attacks, 153–154, 208, 292
TLS (Transport Layer Security), 39, 88, 140, 142, 159, 255
  ClientHello, 257, 264–267
  and Diffie–Hellman, 229
  downgrade protection, 266–267
  handshake, 257, 258–268
  history of, 257
  RC4 in, 102, 104–106
  record, 262
  record payload, 262
  record protocol, 257, 262
  security, 256, 268–271
  ServerHello, 257, 264–267
  session resumption, 267–268
  single round-trip handshake, 267
  version 1.3, 265–267
  zero padding, 263
TLS Working Group (TLSWG), 271
TMTO (time-memory trade-off) attacks, 21, 48, 101
TOFU (trust-on-first-use), 262
traffic analysis, 263
Transport Layer Security. *See* TLS
trapdoor permutations, 196, 197, 199
trapdoors, 196, 197–199
traveling salesman problem, 184
triple DES (3DES), 67, 82–83
trusted third party, 258
trust-on-first-use (TOFU), 262
Turing Award, 216
tweakable encryption (TE), 20–21

U

UDP (User Datagram Protocol), 257
unforgeability, 140
uniform distribution, 27

unitary matrix, 279
universal hash functions, 148–149
Unix, 34
unpredictability, 117
upper bound, 46

V

Vandewalle, Joos, 137
van Oorschot, Paul C., 137
Vigenère, Blaise de, 5
Vigenère cipher, 5–6
virtual private network (VPN), 104

W

Wagner, David, 39, 41, 64, 112
Wegman–Carter MAC, 149–150, 155
Weierstrass form, 232
WEP (Wireless Encryption Protocol), 102, 103–104
Wiener, Michael, 57, 137, 213
Wi-Fi, 87, 103–104
Wilcox-O'Hearn, Zooko, 133, 135
Windows, 37–38
Winnerlein, Christian, 133
Winternitz one-time signature (WOTS), 288–289
Wireless Encryption Protocol (WEP), 102, 103–104
WPA2, 174
Wustrow, Eric, 40, 251

X

Xbox, 137
XOR swap, 112–113

Y

Yao, Andrew C., 230
Yarrow, 30
Yung, Moti, 305

Z

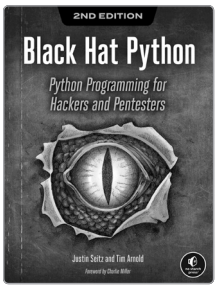
zero-knowledge proof (ZKP), 323–324
  noninteractive, 326
0-RTT data, 267
Zhao, Yunlei, 230
zkSNARK, 327–329
ZUC, 102

Serious Cryptography, 2nd Edition, is set in New Baskerville, Futura, Dogma, and TheSansMono Condensed.

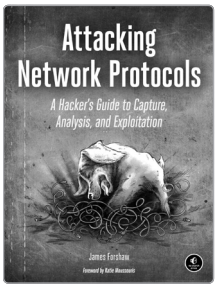
RESOURCES

Visit *https://nostarch.com/serious-cryptography-2nd-edition* for errata and more information.

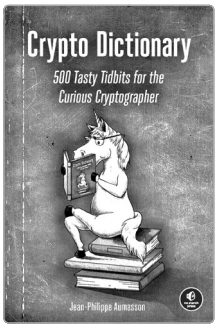
More no-nonsense books from **NO STARCH PRESS**



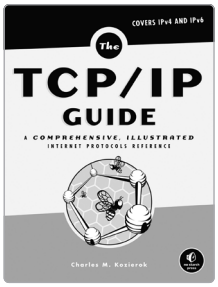
BLACK HAT PYTHON, 2ND EDITION
Python Programming for Hackers and Pentesters
BY JUSTIN SEITZ *AND* TIM ARNOLD
216 PP., $44.99
ISBN 978-1-7185-0112-6



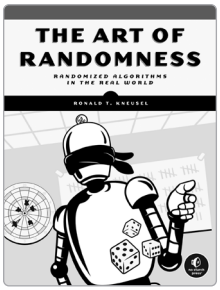
ATTACKING NETWORK PROTOCOLS
A Hacker's Guide to Capture, Analysis, and Exploitation
BY JAMES FORSHAW
336 PP., $49.95
ISBN 978-1-59327-750-5



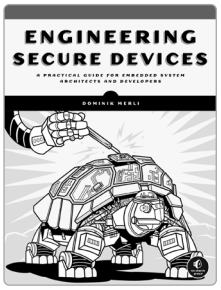
CRYPTO DICTIONARY
500 Tasty Tidbits for the Curious Cryptographer
BY JEAN-PHILIPPE AUMASSON
160 PP., $24.99
ISBN 978-1-7185-0140-9



THE TCP/IP GUIDE
A Comprehensive, Illustrated Internet Protocols Reference
BY CHARLES M. KOZIEROK
1,616 PP., $99.95
ISBN 978-1-59327-047-6



THE ART OF RANDOMNESS
Randomized Algorithms in the Real World
BY RONALD T. KNEUSEL
400 PP., $49.99
ISBN 978-1-7185-0324-3



ENGINEERING SECURE DEVICES
A Practical Guide for Embedded System Architects and Developers
BY DOMINIK MERLI
288 PP., $59.99
ISBN 978-1-7185-0348-9

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

"A thorough and up-to-date discussion of cryptographic engineering, designed to help practitioners who plan to work in this field do better."

—MATTHEW D. GREEN, PROFESSOR, JOHNS HOPKINS UNIVERSITY INFORMATION SECURITY INSTITUTE

This thoroughly revised and updated edition of the bestselling introduction to modern cryptography breaks down fundamental mathematical concepts without shying away from meaty discussions of how they work. In this practical guide, you'll gain immeasurable insight into topics like authenticated encryption, secure randomness, hash functions, block ciphers, and public-key techniques such as RSA and elliptic curve cryptography.

You'll find coverage of topics like:

- The basics of computational security, attacker models, and forward secrecy
- The strengths and limitations of the TLS protocol behind HTTPS secure websites
- Quantum computation and post-quantum cryptography
- How algorithms like AES, ECDSA, Ed25519, Salsa20, and SHA-3 work
- Advanced techniques like multisignatures, threshold signing, and zero-knowledge proofs

Each chapter includes a discussion of common implementation mistakes using real-world examples and details what could go wrong and how to avoid these pitfalls. And, true to form, you'll get just enough math to show you how the algorithms work so that you can understand what makes a particular solution effective—and how they break.

NEW TO THIS EDITION: This second edition has been thoroughly updated to reflect the latest developments in cryptography. You'll also find a completely new chapter covering the cryptographic protocols in cryptocurrency and blockchain systems.

Whether you're a seasoned practitioner or a beginner looking to dive into the field, *Serious Cryptography* will demystify this often intimidating topic. You'll grow to understand modern encryption and its applications so that you can make better decisions about what to implement, when, and how.

About the Author

Jean-Philippe (JP) Aumasson holds a PhD in cryptography and has authored over 60 research articles. His cryptographic algorithms BLAKE2 and SipHash run in every Linux system, with his BLAKE3 renowned for its exceptional speed. Aumasson, currently CSO at Taurus SA, also conducts cryptography training and speaks at leading conferences worldwide such as Black Hat, DEF CON, Troopers, and Infiltrate.

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com